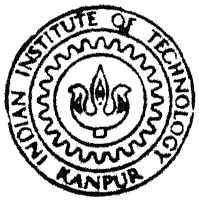
DESIGN AND DEVELOPMENT OF X.25 INTERFACE FOR PC

by

I. RAVI



DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

MARCH, 1989

DESIGN AND DEVELOPMENT
OF
X.25 INTERFACE FOR PC

A Thesis Submitted

In Partial Fulfilment of the Requirements

for the degree of

Master of Technology

by

I. Ravi

to the

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

March 1989

EE-1989-M-RAV-DES

CERTIFICATE

Certified that the work entitled **'DESIGN AND DEVELOPMENT OF X.25 INTERFACE FOR PC'** has been carried out under our supervision and has not been submitted elsewhere for a degree.

Dr. R.N. Biswas
Professor

Dr. K. R. SRIVATHSAN
Assistant Professor

Department of Electrical Engineering
Indian Institute of Technology
KANPUR - 208 016

ACKNOWLEDGMENTS

I express my deepest sense of gratitude to **Dr.K.R.Srivathsan** and **Dr.R.N.Biswas** for their guidance and encouragement during the course of the present work. Working under them has been a rewarding experience.

My thanks to Mr.G.N.M.Sudhakar, Mr.Bhatnagar, Mr.George Joy, Mr.Gopal Krishna, and Mr.Ravi Kumar (REs) of Microprocessor Development Systems Lab. for their help during the present work.

I would like to thank all my friends for the help they provided. My special thanks to Subramaniam, Deepak, and Prasad for their help at critical junctures.

I also would like to thank Mr.Sanjay Kumar for his help in preparing this thesis.

I.Ravi

ABSTRACT

The present work deals with the Design and Development of X.25 interface for PC. The lower two layers of the X.25, i.e. the Physical layer and the Data link layer (LAPB) are implemented. The physical layer has been implemented by the hardware, based on the up8088 and the multiprotocol serial controller (MPSC) 8274. The hardware is designed to be a general purpose data communication hardware. The data link layer (LAPB) is implemented by the MPSC-8274 and the firmware. A major part of the firmware is developed in 'C' language. The X.25 interface board communicates with the PC through the PC-Bus, and forms an I/O port of the PC.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

The 1970s heralded the beginning of the wide area networks (i.e. Public data networks) and since then both the number and coverage of these networks has increased. As a result, the necessity for uniformity of procedures for user to network interface and internetworking forced the international bodies to come up with recommendations of protocol standards. In 1974 CCITT issued the first draft of X.25 standards, defining the user to network interface. It was revised in 1976, 1980 and 1984 [1].

With the introduction of low cost PCs in 1980, the worldwide usage of PCs has increased. When connected to wide area networks they become a valuable tool in data communications and have an access to the globally situated mainframes and databases. Keeping this in view, the present work attempts to develop X.25 interface for the PC.

1.2 WIDE AREA NETWORKS

The major characteristics of the wide area networks are :

1. They employ mixed channels
2. Channels are slow (=< 64 Kbps) , and are error prone (typically $1:10^{4}$ to $1:10^{6}$)

3. Users and Packet switching exchanges are globally situated.

The wide area networks are classified into two types [4], namely :

1. Connection oriented (also called virtual circuit service) networks
2. Connection less (also called datagram) networks.

A connection oriented network is one in which a logical connection exists between the communicating users. Before communicating through the network, the users establish a logical connection through a call negotiation procedure. During this procedure the complete destination address is specified. The message packets do not carry the complete destination address. After the completion of data transfer the users perform the connection release procedure. The connection oriented network provides network wide acknowledgments and performs error recovery, to assure that the user's data is not lost in the network. It assures that the data is delivered in the same sequence as it enters the network. The call negotiation and call release procedures present additional overhead. This overhead is considerable for single packet messages. But it is useful for multiple packet messages, as the complete destination address need not be specified in every packet.

The connection less network does not require the call establishment for data transfer. The communicating users are not logically connected and every message packet carries the complete destination address. It does not support the network wide acknowledgments and error recovery procedures. The sequence in which the data enters this network is not necessarily preserved upon delivery at the destination. As there are no network wide acknowledgments, the data may be lost in the network. As a result of the minimal functions it supports, the overhead is low. This network is suitable for transaction oriented communications, where the data volumes are low. However, the overhead is high for large volume data transfers as every transmitted packet has to carry the complete destination address.

The connection less networks were popular between 1970 and 1977. In 1976, CCITT X.25 standards set the direction towards connection oriented networks [4]. The X.25 is developed as a layered model for public data networks, supporting the lower three layers of the seven layer OSI reference model. These three layers are the physical layer, the data link layer, and the network layer. The physical layer provides the physical connectivity between the network and the user terminal. The data link layer assures an error free data channel, inspite of the error prone nature of physical channel. Through statistical multiplexing techniques the network layer supports upto 4095 logical

connections over a single physical channel. The X.25 has elaborate flow control mechanisms to regulate the data traffic to and from the network. In addition it provides a number of end user facilities [2.3.2].

The CCITT X.25 is accepted worldwide as the user-network interface standard, for wide area networks. Presently a number of countries have developed the X.25 networks. The various the wide area networks being developed in India are given in Table 1.1. With a few enhancements the X.25 procedures are adopted in the Integrated Services Digital Network (ISDN) procedures [1].

1.4 ORGANIZATION OF THE THESIS

The aim of this thesis is to design and develop an X.25 interface for the PC. The organization of the thesis is as given in below :

Chapter 2 summarizes the features of CCITT X.25 standards.

Chapter 3 presents an overview of the hardware and firmware implementation aspects of the board.

Chapter 4 discusses the implementation details of the hardware.

Chapter 5 discusses the implementation details of the firmware.

Chapter 6 concludes the work with a few suggestions for further work.

Table 1.1

Wide area networks developed/under development in India

Network	Developed by
INDONET	Computer Maintainance Corporation (CMC)
OILCOMNET	Indian Oil Industry
NICNET	National Informatics Centre (NIC)
SAILNET	Steel Authority of India Ltd. (SAIL)
COALNET	Coal India Ltd.
RESNET	Defence Research Development Organization (DRDO)
Remote Area Business message network (RAMN)	Department of Telecommunications (DOT)
ERNET	Educational and Research establishments Department of Electronics (DOE)
PSTN VIKRAM	Department of Telecommunications (DOT)

CHAPTER 2

CCITT X.25 RECOMMENDATIONS

CCITT X.25, defining the DTE and DCE interface, is independent of the internal structure of the packet switching network. Though it defines DTE-DCE interface only, it has end to end significance as the data has to be routed end to end. The physical location of X.25 interface in a network is shown in Fig.2.1. The X.25 interface defines three levels of interface viz. physical level, data link level and network level as shown in Fig.2.2.

2.1 PHYSICAL LEVEL INTERFACE [2]

The physical characteristics and the Call Control Procedures for an interface between DTE and DCE, at physical level, are defined by X.21. The X.25 employs the Leased Circuit service as defined in X.21 [2]. This interface consists of a duplex, synchronous and point-to-point physical interchange circuits. The electrical characteristics of the interchange circuits are defined by X.26/X.27 [5]. The DTE/DCE interface connector and the pin assignment are according to ISO4903 [2,5].

The interchange circuits of X.21 are shown in Fig.2.3. The T and R circuits convey data and control information, while C and I circuits function similar to "ON/OFF hook"

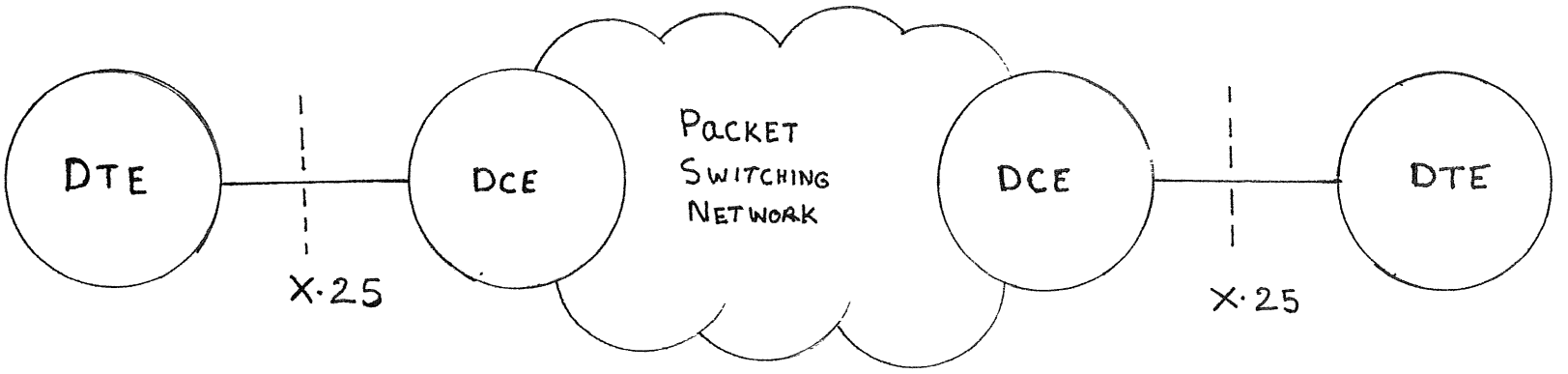


Fig. 2.1 X.25 INTERFACE

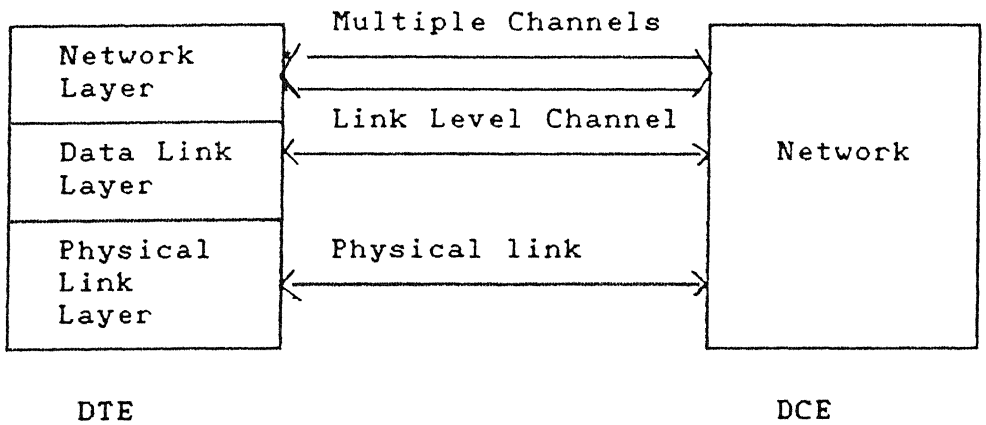


Fig. 2.2 X.25 Layers

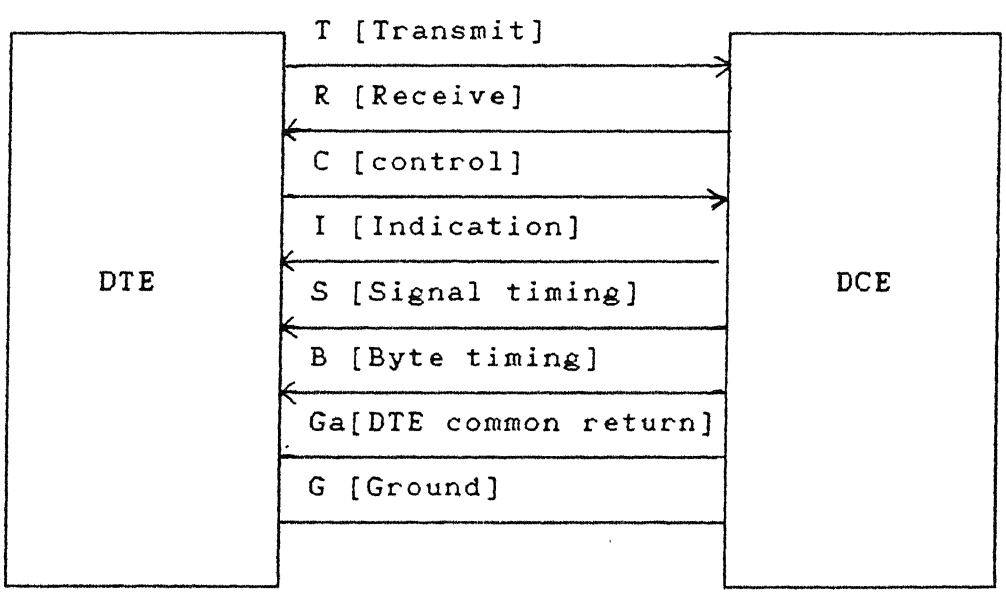


Fig. 2.3 X.21 Interchange circuits

indication. The S circuit provides signal element (bit) timing from DCE, and optionally in some networks the B circuit provides an octet alignment timing.

The X.21 interface can be in any one of the following phases:

i) Quiescent phase

ii) Packet switched service phse.

The state diagrams of these phases are shown in Fig. 2.4 and Fig. 2.5.

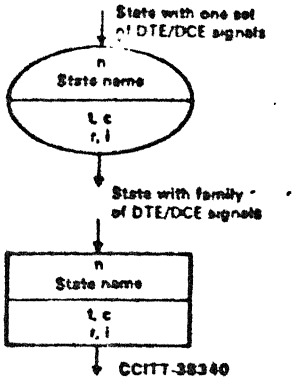
CCITT has also defined an interim standard X.21bis, at physical level interface, to accommodate the existing synchronous V series DTEs. The interchange circuits of X.21bis are given in Table 2.1. The X.21 DTE/DCE and X.21bis DTE/DCE can interwork [2].

2.2 DATA LINK LEVEL [1,3]

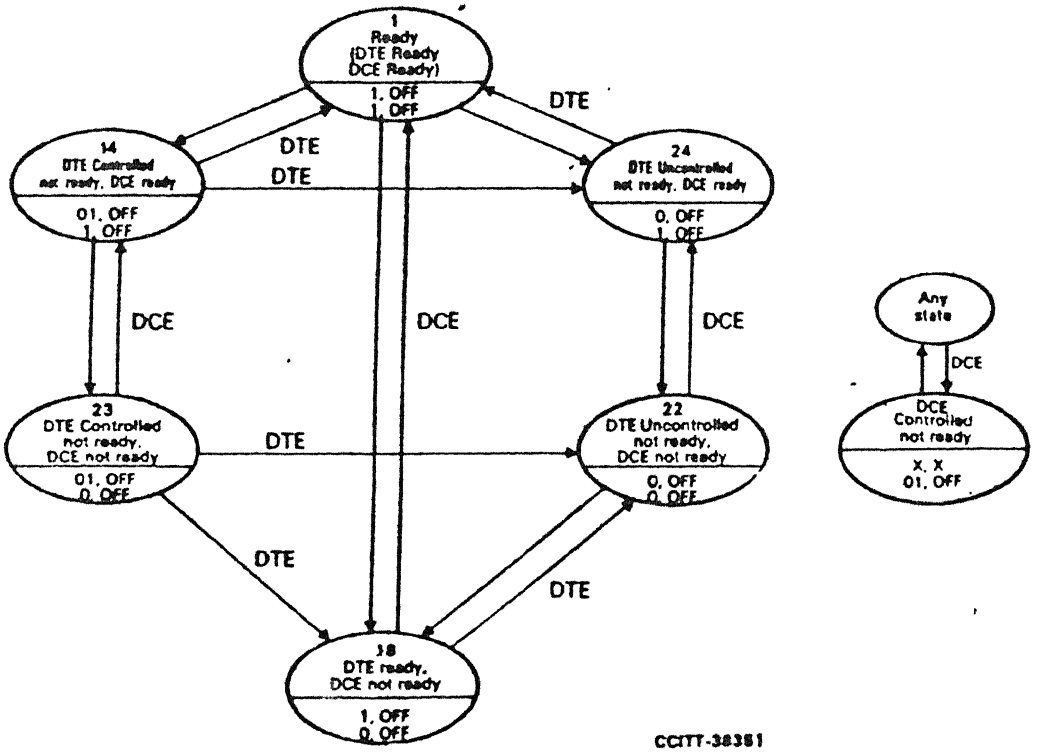
The data link level interface is defined by Link Access Procedure-Balanced (LAPB). The use of Link Access Procedure (LAP) is permitted, but that of LAPB is encouraged. Through a system of acknowledgment, error detection and retransmission, LAPB, a subset of HDLC, provides a practically error-free data channel despite the unreliability of the physical medium. The data at the receiving end is delivered without loss, duplication and in the same sequence. The basic transmission unit at this level is the frame.

Interface signalling state diagrams

Definition of symbols used in the state diagrams



n	State number
t	Signal on T circuit
c	Signal on C circuit
r	Signal on R circuit
i	Signal on I circuit
T	Transmit interchange circuit
C	Control interchange circuit
R	Receive interchange circuit
I	Indication interchange circuit
D	DTE or DCE data signals
0 and 1	Steady binary conditions
01	Alternate binary 0 and binary 1
X	Any value
OFF	Continuous OFF (binary 1)
ON	Continuous ON (binary 0)
IA5	Characters from International Alphabet No 5 (Recommendations V.3 and X.4)
+	IA5 character 2/11
BEL	IA5 character 0/7
SYN	IA5 character 1/6
	Transition with indication of whether DTE or DCE is responsible for transition



Note 1 – This state diagram shows transitions that will be allowed by all Administrations. Other transitions are possible and may be allowed by some Administrations.

Note 2 – *DCE controlled not ready* appearing during the call establishment phase should be interpreted as a *DCE clear indication*.

Fig. 2-4 **Quiescent states**

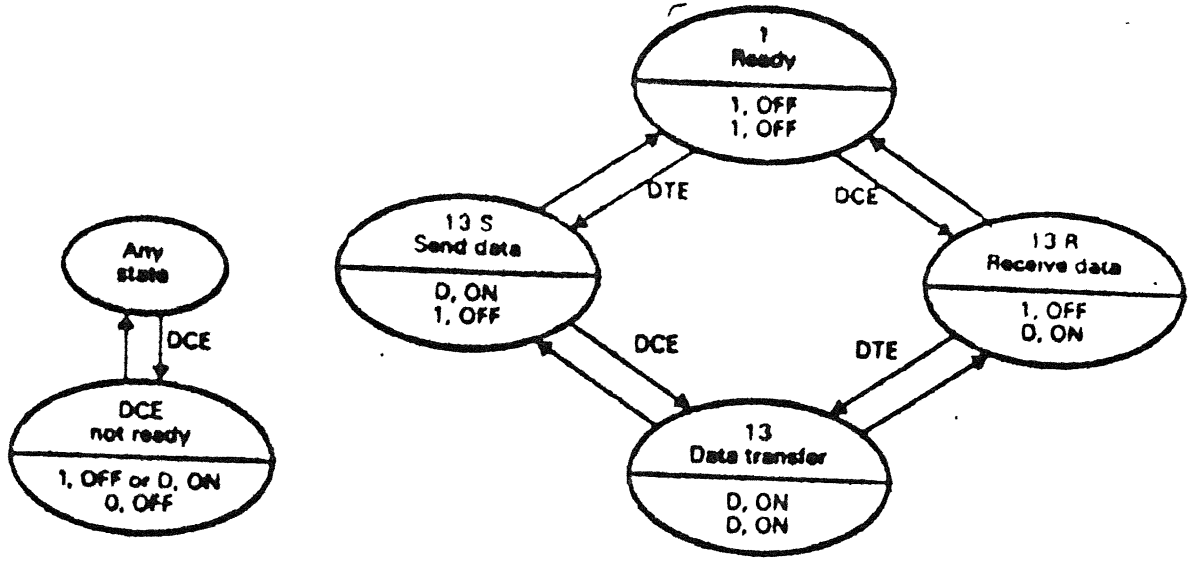


Fig 2.5 Leased circuit service – point-to-point and packet-switched service

TABLE 2-1

V.24 Interchange circuit No.	Designation
102	Signal ground or common return
103	Transmitted data
104	Received data
105	Request to send
106	Ready for sending
107	Data set ready (see Note 1)
108/1	Connect data set to line (see Notes 2 and 3)
109	Data channel received line signal detector
114	Transmitter signal element timing (DCE) (see Note 4)
115	Receiver signal element timing (DCE) (see Note 4)
140	Loopback/maintenance test (see Note 5)
141	Local loopback (see Note 5)
142	Test indicator (DCE)

Note 1 – Circuit 107 shall go OFF only in cases of DCE power-off (normally the indeterminate state is interpreted as OFF), loss of service (see § 3.2 below) or when circuit 108/1, when implemented, is turned OFF.

Note 2 – Not required for V.29, V.35 and V.36 compatible interfaces.

Note 3 – The DCE interprets the ON condition on circuit 108/1, when implemented, as an indication that the DTE is operational. If circuit 108/1 is not provided the DCE will consider the lack of circuit 108/1 as the ON condition. The DCE turns circuit 107 ON while circuit 108/1, if present, is ON and the circuit connection is available.

Note 4 – The DCE shall provide the DTE with transmitter and receiver signal element timings; this is done by feeding circuits 114 and 115 with the same timing signal from the DCE.

Note 5 – Not required in those networks which do not provide automatic activation of the loops.

The LAPB frame structure is shown in Fig.2.6. the frames are classified into Information (I), Supervisory (S) and Unnumbered (UN) frames. The frame type is determined by the control field. The different types of frames used in LAPB and their control field formats are shown in Table 2.2. The address field identifies whether the frame is a command or a response. If the frame is a Command, then it's address field has the destination address, while a Response frame has the source address. The P/F bit of the control field serves a function in both command and response frames. It is referred as P-bit in Command frames and as F-bit in Response frames. Command frames with P-bit set are sent to solicit response at the earliest opportunity with F-bit set.

To ensure that frames are received in sequence, sequence numbers V(s), N(s), V(r) and N(r) are used in LAPB. These sequence numbers are defined as follows:

Send State Variable V(s)

It denotes the sequence number of next in-sequence 'I' frame to be transmitted. V(s) can take values from 0 to 7 (modulo 8), provided it does not exceed the receive sequence number N(r) by more than the maximum number of outstanding 'I' frames. V(s) is incremented by one with each successive 'I' frame transmission.

Flag (8)	Address (16)	Control (8)	Information	CRC (16)	Flag (8)

Fig 2.6 LAPB Frame structure.

Table 2.2

LAPB frame types

Frame Type	Command	Response	Control Field Encoding 1 2 3 4 5 6 7 8
Information (I)	I	-	0 N(S) P/F N(R)
Supervisory (SS)	RR (Receive ready)	RR	1 0 0 0 P/F N(R)
	RNR (Recev. not rdy.)	RNR	1 0 1 1 P/F N(R)
	REJ (Reject)	REJ	1 0 0 1 P/F N(R)
Unnumbered (UN)	SABM (Set asynchronous balanced mode)	-	1 1 1 1 P 1 0 0
	DISC (Dicon-nect)	-	1 1 1 1 P 0 1 0
		UA (Un-numbered acknowledgement)	1 1 0 0 F 1 1 0
		FRMR (Frame reject)	1 1 1 0 F 0 0 1

Send Sequence Number N(s)

It is contained in 'I' frames only and denotes the sequence number of the frame being transmitted.

Receive State Variable V(r)

It denotes the sequence number of next in-sequence 'I' frames to be received. It can take values from 0 to 7 (modulo 8). The value V(r) is incremented by one with the receipt of an error free in-sequence 'I' frame whose send sequence number N(s) matches the receive state variable V(r).

Receive Sequence Number N(r)

All 'I' and 'S' frames contain N(r), the expected sequence number of the next received 'I' frame. Prior to the transmission of a frame, the value of N(r) is set equal to the current value of receive state variable V(r). N(r) indicates to the transmitting DTE or DCE that the receiving DTE or DCE has correctly received all 'I' frames upto and including (N(r)-1).

Supervisory frames of LAPB are used to acknowledge the receipt of 'I' frames, to indicate "busy" or "clear busy" condition , and reject out of sequence frames. The busy condition arises when the DTE or DCE is temporarily unable to receive 'I' frames due to constraints like non-availability of buffer space. Receive Not Ready (RNR) frames

are transmitted by busy DTE or DCE. Busy condition is cleared by the transmission of Receive Ready (RR), Reject (REJ), Unnumbered Acknowledge (UA) or SABM Commands.

Unnumbered frames are used to establish or disconnect logical link, to report status of DTE or DCE when it is logically disconnected, and to report by Frame Reject (FRMR) frame, an error condition not recoverable by retransmission of identical frames.

2.2.1 Link Establishment Phase

The DCE indicates it's ability to setup link by transmitting continuous flags (active channel state).

On receiving an SABM Command, the DCE returns an UA Response to the DTE and sets it's V(s) and V(r) to 0. If DCE wants to establish the link, it will send the SABM Command and starts timer T1. Upon receipt of UA frame, link is established and DCE resets V(s) and V(r) to 0 and stops timer T1. If timer T1 runs out before the receipt of an UA response, SABM is retransmitted and timer T1 is re-started. After transmission of SABM for N2 times, by DCE, the recovery action will be initiated.

2.2.2 Information Transfer Phase

After having transmitted the UA Response to a received SABM Command or having received an UA Response to a

transmitted SABM Command, the DCE will accept 'I' and 'S' frames according to the procedures described below:

When DCE has an 'I' frame to transmit, it will transmit it with N(s) equal to the current V(s), and N(r) equal to current V(r). Timer T1 is started at the instant of transmission. If the send state variable V(s) is equal to the last value of N(r) plus 7, the DCE will not transmit any new 'I' frames. However, it may retransmit an 'I' frame, if timer T1 runs out or a REJ is received. If an 'I' frame is retransmitted for N2 times, the logical link is Reset.

When DCE receives an 'I' frame with correct Frame Check Sequence (FCS), and sequence number equals to V(r), it will accept the information field of 'I' frame and increments it's local V(r). It will acknowledge the received 'I' frame, by transmitting an 'I' frame (if available) or RR or RNR, with N(r) equal to V(r). When an out of sequence 'I' frame is received, only the received 'I' frame's control field is accepted, and a REJ with N(r) equal to V(r) is sent. Only one REJ exception condition for a given direction of information transfer is allowed at any given time. The REJ exception is cleared on receipt of an 'I' frame with N(s) equal to N(r) of the REJ frame sent. If DCE receives an invalid frame that cannot be corrected by retransmission, a Frame Reject (FRMR) is sent.

However, if DCE is busy it will ignore the information contents of received 'I' frames. The above discussion is also valid at DTE end.

2.2.3 Link Disconnection Phase

This phase can be entered from information phase only. During the information phase DTE will indicate the logical disconnection of the link by transmitting a DISC Command. DCE will then respond through an UA frame to DTE and enters the disconnected phase. DTE will enter disconnected phase on receiving the UA frame.

If DCE wants to disconnect the link, it sends DISC Command and starts timer T1. On receipt of an UA frame from DTE, DCE will stop it's timer T1. If the timer T1 runs out before receiving an UA frame, DISC is retransmitted and timer T1 is restarted. After transmission of DISC Command for N2 times, DCE will initiate recovery action.

2.2.4 Link Resetting Procedure

The link resetting procedure is applicable during information transfer phase only. The DTE or DCE indicates link resetting by transmitting an SABM Command. After receiving an SABM Command, DCE or DTE sends, at the earliest opportunity, an UA frame to the DTE or DCE, and resets it's V(s) and V(r) to 0. This also clears a DCE and/or DTE busy condition, if present. Prior to initiating this procedure,

DTE or DCE may initiate a link disconnect procedure. The DCE may ask DTE to reset link by transmitting DM frame. The link may also be reset by DCE when it receives a DM, or FRMR, or an UA frame or an unsolicited response with F-bit set to 1.

2.2.5 Link Level System Parameters

Timer T1: It protects against the missing acknowledgments. It's value is specified by the network administrator.

Maximum Frame Size N1: depends upon the maximum length of the information fields transferred across the DTE/DCE interface. In the design discussed in later chapters, the value of N1 is selected to be 256 bits.

2.3 NETWORK LEVEL INTERFACE [1,3,4]

X.25 at packet level operates on the premise of virtual circuit services. A virtual circuit (also called a logical channel) is one in which the host perceives the existence of a dedicated physical circuit to the peer host m/c, yet in reality the "dedicated" physical circuit is allowed to multiple users. Through the use of statistical multiplexing techniques, different user packets are interleaved onto one physical channel. X.25 supports upto 4095 virtual circuits and each virtual circuit is identified by a logical group number and channel number.

X.25 provides the following channel options to establish and maintain communications:

1) Permanent Virtual Circuit (PVC)

2) Virtual Call (VC)

3) Fast Select Call

4) Fast Select Call with Immediate Clear

When a permanent virtual circuit is allocated, the transmitting DTE is assured of obtaining a connection to the receiving DTE through the packet network. The PVC requires no call set-up or clearing procedures, and logical channel is continuously in data transfer state. PVC is analogous to a leased line in telephone network.

A virtual call needs a call set-up procedures before the start of data transfer and a call clear procedure at the end of data transfer. The X.25 virtual call process is shown in Fig.2.7.

A fast select call is also similar to a virtual call except that it allows call request packet to contain user data upto 128 octets (bytes). The called DTE is allowed to respond with a call accepted packet, which can contain user data.

Fast select with immediate clear, also allows call request packet to contain user data. This packet is transmitted through the network to the receiving DTE , which upon acceptance, transmits a clear request (which also

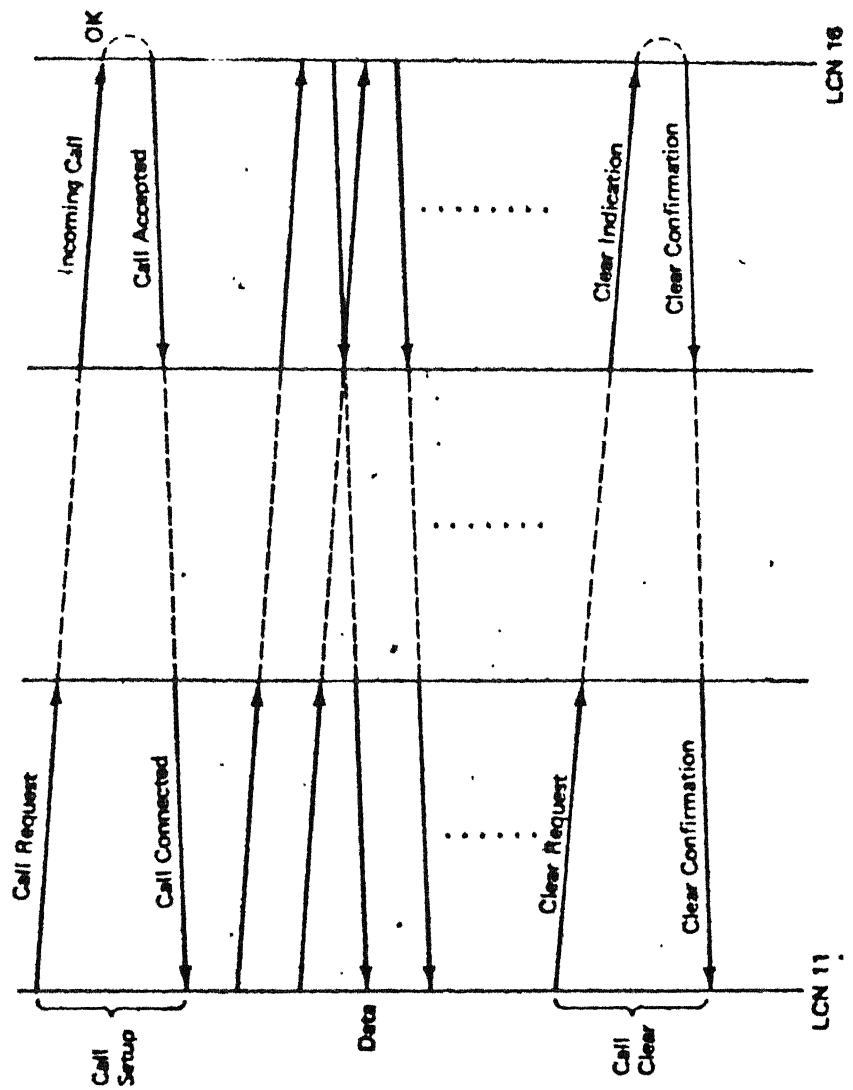


Fig 2.7 VIRTUAL CALL PROCESS

contains user data). The clear request is received at the originating site as a clear indication packet. This site returns a clear confirmation, which cannot contain user data.

Fast select provides support for user applications that have only one or two transactions such as inquiry/response applications. These applications cannot effectively use virtual circuit due to the overhead and delay in call establishment and disconnection.

2.3.1 Packet Types [1,3]

The various packet types implemented in X.25 are shown in Table 2.3.

The interrupt procedure allows a DTE to transmit one non sequenced packet to another DTE without following normal flow control procedure established in X.25. However, an interrupt packet requires an interrupt confirmation before another interrupt packet can be transferred. The use of interrupts does not effect regular data packets with-in VC or PVC. This facility is useful for a situation where an application requires the transmission of data under unusual conditions.

The Receive Ready (RR) and Receive Not Ready (RNR) packets are similar to the same commands in LAPB. They serve the important function of flow control, to limit the rate at

TABLE 2-3 Packet Types

Packet Type		Service	
From DCE to DTE	From DTE to DCE	VC	PVC
Call Set-up and Clearing			
Incoming call	Call request	X	
Call connected	Call accepted	X	
Clear indication	Clear request	X	
DCE clear confirmation	DTE clear confirmation	X	
Data and Interrupt			
DCE data	DTE data	X	X
DCE interrupt	DTE interrupt	X	X
DCE interrupt confirmation	DTE interrupt confirmation	X	X
Flow Control and Reset			
DCE RR	DTE RR	X	X
DCE RNR	DTE RNR	X	X
	DTE REJ	X	X
Reset indication	Reset request	X	X
DCE reset confirmation	DTE reset confirmation	X	X
Restart			
Restart indication	Restart request	X	X
DCE restart confirmation	DTE restart confirmation	X	X
Diagnostic			
Diagnostic		X	X
Registration			
Registration confirmation	Registration request	X	X

VC = Virtual Call PVC = Permanent Virtual Circuit

which DTE or DCE receives packets, and acknowledges the received packets. The transmission of RR or RNR packets has an end-to-end effect, to prevent excess traffic from entering into the network. X.25 provides individual flow control for each logical channel.

The Reject (REJ) packet specifically rejects the received packet. If it is used, the receiving station requests retransmission of packets with the count beginning with the count in the packet receive sequence field.

The Reset packets are used to re-initialize a switched virtual call or permanent virtual call. The Reset procedure removes in each direction, between the two stations (for one logical session), all data and interrupt packets which may be in the network. Reset procedures are necessary when problem conditions arise, such as lost packets, duplicate packets, or packets that cannot be re-sequenced properly. It can be initiated, either by DTE or DCE, only during data transfer state.

The restart procedure is also like Reset procedure except that it reinitializes all the 4095 logical channels and all outstanding packets are lost. It is used when a severe problem, like network crash, occurs. When a DTE sends Restart, the network transmits a Restart to each DTE that has a virtual circuit session with DTE that issued Restart.

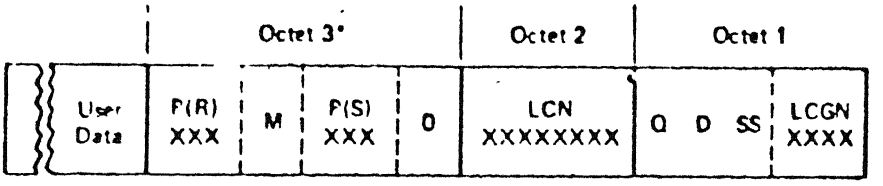
The clear packet is mainly used to clear a DTE-DCE virtual circuit session. It is also used to indicate that a call request cannot be completed, in which case the fourth octet of the packet contains a reason for the clear.

2.3.2 Packet Formats [1]

Packets are basically classified into Data Packets and Non-data Packets. The length of a user data field in a data packet is 128 bytes or octets, but X.25 provides options for octet lengths, which are 16, 32, 64, 256, 512, 1024, 2048 and 4096 octets. If the user data field exceeds network permitted maximum field, the receiving DTE will reset the virtual circuit by issuing a reset packet.

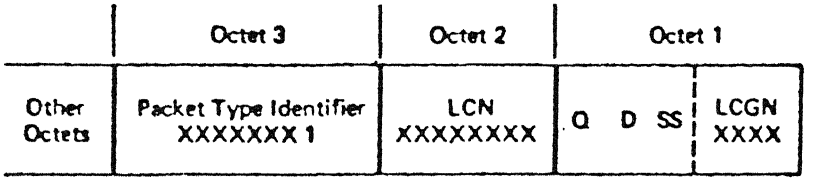
Every packet transferred across the DTE/DCE interface must contain atleast three octets, constituting the packet header. The packet formats for data and non-data packet formats are shown in Fig.2.8.

The last four bits of the first octet contain the General Format Identifier (GFI). The bits 5 and 6 of GFI (i.e. SS),are used to indicate the sequencing for the packet session, as X.25 allows two sequencing options. The first option is modulo 8, which permits sequencing options from 0 to 7, while the second option is modulo 128, which permits sequencing options from 0 to 127. The seventh bit (D-bit), when 0 only network acknowledgments are provided,and when 1 DTE to DTE acknowledgments are provided. The eighth (Q-bit)

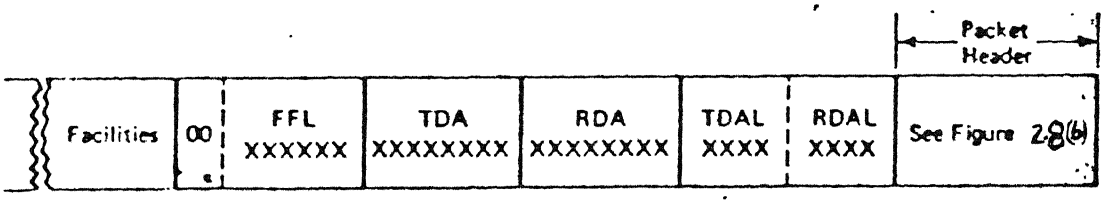


*: Modulo 128 Uses a Fourth Octet for Extended Sequencing
SS = 01 for Modulo 8
SS = 10 for Modulo 128

(a) Data Packet Header



(b) Nondata Packet Header



(c) Nondata Packet

P(R): Receiving Sequence Number
M: Packet Category Indicator
P(S): Sending Sequence Number
LCN: Logical Channel Number
Q: Qualifier Bit
D: Delivery Confirm Bit
SS: Modulo Bits
LCGN: Logical Channel Group
FFL: Facilities Field Length
TDA: Transmitting DTE Address
RDA: Receiving DTE Address
TDAL: Transmitting DTE Address Length
RDAL: Receiving DTE Address Length

Fig 2.8 X.25 Packet Formats

is normally set to 0. It is significant in Packet Assembler Disassembler (PAD) where a packet received with Q-bit set to 1 indicates a PAD Control Packet, while a packet with Q-bit set to 0 indicates User Data Packet. The logical channel number (second octet) together with logical group number identifies the logical channel. The first bit of third octet when 0, indicates data packet else non-data packet. The content of octet 3 of the packets is shown in Fig. 2.8. The M-bit (i.e. More data) of third octet of data packets identifies a related sequence of packets traversing through the network. This capability aids the network and DTEs in preserving the identification of blocks of data when network divides these blocks into small packets.

The additional fields inside X.25 non-data packet are shown in Fig.2.8(c). For call establishment packets, the DTE address and address lengths are included. The addressing convention is defined by X.121. After a call is established, network uses the associated logical channel number to identify DTE to DTE session. The facility field of non-data packets may be used in the event the DTEs wish to use options provided in X.25 standard. Once the value of the facility has been established, it remains in effect until the call is cleared.

X.25 provides a number of optional features [1], of which some of are listed below:

1) Incoming calls barred/Outgoing calls barred
2) One way logical channel outgoing/One way logical channel incoming
3) Selection of throughput rate
4) Flow control parameter negotiation
5) Closed user groups to provide a measure of security/privacy in public data networks
6) Reverse charging
7) Local charging prevention
8) Hunt group
9) Call redirection facility

CHAPTER 3

X.25 INTERFACE FOR PC

The X.25 Interface for PC is based on the development of a printed circuit board that can be interfaced to the PC. The board supports the lower two layers, i.e. the physical layer and the data link layer(LAPB), of the three layered X.25 protocol. The board development can be classified into:

1. Hardware development,
2. Firmware development.

An overview of the hardware and firmware development is discussed in this chapter. The implementation details of hardware are discussed in chapter 4 and those of firmware in chapter 5.

3.1 BLOCK LEVEL DESCRIPTION OF HARDWARE

The hardware of 'X.25 Interface for PC' supports a X.21 port, an RS-232 port, I/O interface to PC and a part of data link layer. The block diagram of the hardware is shown in Fig.3.1. The different blocks,are :

1. Clock circuitry
2. Processor, and address,data buffering logic
3. Wait state logic
4. Memory

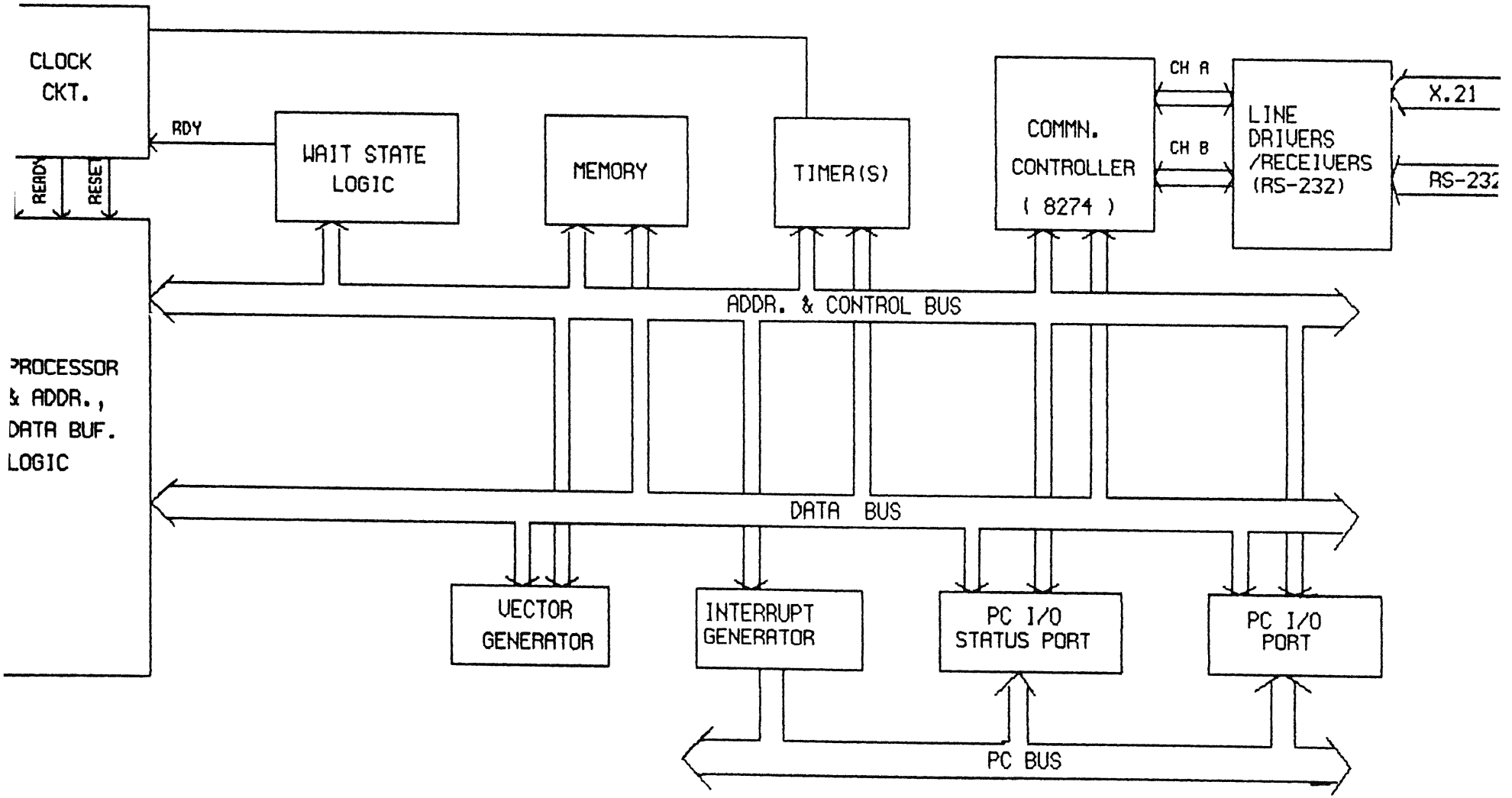


Fig 3.1 Block diagram of hardware

5. Timer(s)
6. Serial controller and Line receivers/drivers
7. PC I/O port
8. PC I/O status port
9. Interrupt generator
10. Vector generator

The general features of these blocks are given below :

1. The Clock circuit provides clock to the processor, the Serial controller and the timer(s). At Power on it resets the Processor, and Serial controller. It accepts 'asynchronous ready' from the wait state logic and provides 'synchronous ready' to the processor.
2. The Processor on board is uP 8088 [8]. The Processor's address-data, address-status signals are demultiplexed and buffered. The address lines are decoded to provide Chip-Selects to addressable devices, on board.
3. For timing compatibility during data transfer between the Processor and Serial controller/Timer(s), atleast one wait state has to be inserted in the read/write access to them. The on board wait state logic inserts these wait states.

4. An 8253/8254 [8], with three Timers, is used for maintaining the timings required for the software time outs.
5. The memory, on board, can vary from 8k ROM and 8k RAM to 32k ROM and 64k RAM. The firmware requires a minimum of 6k ROM and 8k RAM.
6. The Multipurpose Serial Controller (MPSC-8274) [7] supports two independent serial I/O channels, channel A and channel B. Channel A is a used to support X.25 and provides the X.21 interface. The X.21 signals are level translated (TTL/RS-232c) using Line drivers/receivers. The MPSC is programmed in HDLC/SDLC mode. In this mode, it simplifies the implementation of LAPB, as it provides the following features :

    i. Transmission of flags
    ii. Bit insertion, and deletion of inserted bits
    iii. CRC checking
    iv. Facility to abort frame transmission.

   The MPSC-8274 is programmed in 8086/8088 interrupt vector mode. Internal to MPSC there are multiple sources of interrupts and the MPSC itself resolves the priorities of these interrupts and activates the interrupt line.
7. The data communication between PC and X.25

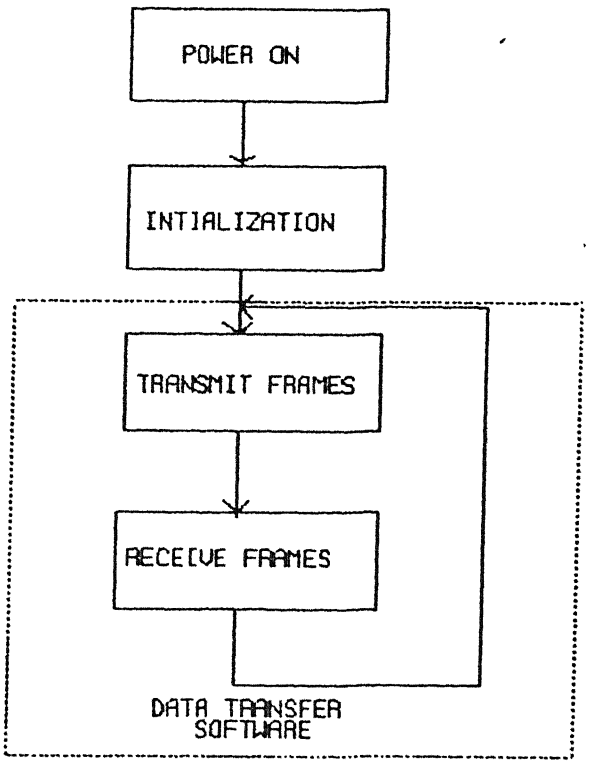
board is through the PC I/O port [9]. This bi-directional I/O port is based on two octal latches. The status of the PC I/O port (i.e.,port full/port empty) and card busy/not busy, are available in PC I/O status port. Whenever PC writes data to the PC I/O port, the interrupt generator interrupts the on board Processor. This interrupt is daisy-chained to the MPSC interrupt. During the interrupt acknowledge cycle of the this interrupt, the Vector generator places the vector on the data bus.

8. When the board writes a character to the PC I/O port, the interrupt generator activates the IRQ3 line of PC. The PC processor as well as the board Processor, poll the status port before writing to the PC I/O port.

3.2 BLOCK LEVEL DESCRIPTION OF FIRMWARE

The firmware, on board, implements the LAPB protocol. A major part of this firmware is developed in 'C' language [10,11,12]. The I/O interface routines are developed in Assembly language [12]. The major blocks of this firmware are shown in Fig.3.2. The firmware can be classified as :

1. Initialization software
2. Data transfer software
3. Interrupt service routines



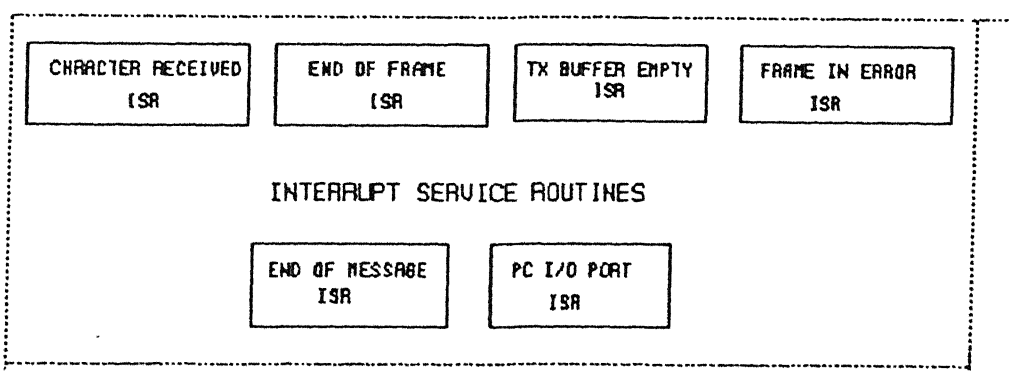


Fig. 3.2 Major blocks of software

3.2.1 Initialization Software

At power on, the control is transferred to the initialization routine. The major functions of this routine are, as follows:

1. Initialize all I/O devices, vector table and data variables,
2. Program the MPSC-8274 in HDLC/SDLC mode [6,12],
3. Reset spurious interrupts and enable interrupts,
4. Establish the balanced data link.

3.2.2 Data transfer Software

After initialization, Firmware is ready for data transfer and control is transferred to data transfer routine. The major functions of this routine are :

1. Assemble and transmit frames,
2. Analyze received frames,
3. Transfer and receive data to/from PC.

The receive modules, of the data transfer routine, accept only those frames which are error free and of size less than the maximum frame length. Depending on the type of frame received the actions taken are, as described below :

1. I-Frame : If the frame's Send sequence number N(S) matches with local Receive state variable V(R), the information content of the frame is transferred to the receive packet buffer. Later, the data in

receive frame buffer is transferred to PC. Incase of a mismatch a reject (REJ) is transmitted.

2. RR/RNR Frame: On receiving a RR or RNR frame the status of destination is noted. If the received frame is a 'RNR' frame, transmission is suspended till an 'RR' frame is received.

3. REJ Frame : On receiving a 'REJ' frame the Send state variable V(S) is reset to the receive sequence number N(R), and all the unacknowledged frames are retransmitted.

4. UN Frame : On receiving a 'SABM' frame, V(S) and V(R) are reset to zero and, the balanced link is re-established. If the received frame is 'DISC' or 'DM' frame, the link is disconnected. A received 'UA' frame is an acknowledgment for the previously transmitted 'SABM' or 'DISC' frame.

If the received frame's P bit is set, an appropriate response frame with F bit set is transmitted.

The data received from PC is loaded into transmit packet buffer. 'I' frames of this data are assembled and transmitted across link. A transmitted 'I' frame is not discarded from transmission window till it is acknowledged. With the transmission of each 'I' frame a software timer is

started. If the timer expires before it is acknowledged the 'I' frame is retransmitted.

3.2.3 Interrupt Service routines

The data transfer to and from MPSC-8274 [7], for serial I/O, and data reception from PC are interrupt driven. The various sources of interrupts, on board, are :

1. Character reception,
2. Transmit buffer empty,
3. Transmit under run
4. End of frame
5. Received frame in error (i.e., receive overrun, aborted frame)
6. PC I/O port

When a character is received, the receive character interrupt service routine (ISR) is invoked. The received character is read and transferred to receive frame buffer. On receiving a complete frame, the end of frame (EOF) interrupt occurs and 'EOF' ISR is invoked. In this ISR the CRC error is checked and if found to be erroneous, the frame is discarded. On receiving error free frame the receive frame buffer's pointers are updated.

If the transmit buffer becomes empty, during a frame transmission, the transmit buffer empty interrupt occurs and

invokes the transmit ISR. In this ISR the next character of the frame is loaded into the transmit buffer. If there are no more bytes of the frame to be transmitted the transmit buffer empty interrupt is reset. On resetting this interrupt, the transmitter underruns and the transmit underrun/end of message (EOM) interrupt arises. On transmitter underrun the CRC bytes and closing flag are transmitted [7].

The 'EOM' interrupt invokes 'EOM' ISR, in which the transmit frame buffer pointers are updated. Also the first character of next frame, if available, is loaded into the transmit buffer, to re-enable the transmit buffer empty interrupt. When a frame in error interrupt occurs, it is reset by writing error reset/reset external status commands [7] to MPSC-8274.

The routine invoked by PC I/O port interrupt reads the PC I/O port and loads the character in transmit packet buffer. If the firmware can not accept any more data from PC, due to buffer space limitations, it sets the CARD_BUSY bit of PC I/O status port. This bit is cleared when firmware is ready to receive more data from PC.

CHAPTER 4

HARDWARE IMPLEMENTATION

This chapter describes the hardware of the X.25 interface. The hardware is interfaced to the PC through the PC-Bus, to ensure high speed data transfer between the interface board and the PC. The process on the board runs independent of the process on the PC, reducing the load on the PC. The hardware, of the board is discussed in two sections, namely:

1. X.25 Circuitry
2. PC interface Circuitry.

4.1 X.25 CIRCUITRY

The X.25 circuit is shown in Fig. 4.1 to Fig. 4.14. This circuit is based on the CPU-uP8088 and the MPSC-8274, and is designed to work up to 8 Mhz. The CPU and it's clock circuit are shown in Fig.4.1. The diode D1 ensures the fast discharge of the capacitor C1 'at power off'. The address data lines are de-multiplexed, and buffered as shown in Fig. 4.2. For de-multiplexing, the address is latched by IC4 and IC5, at the falling edge of ALE.

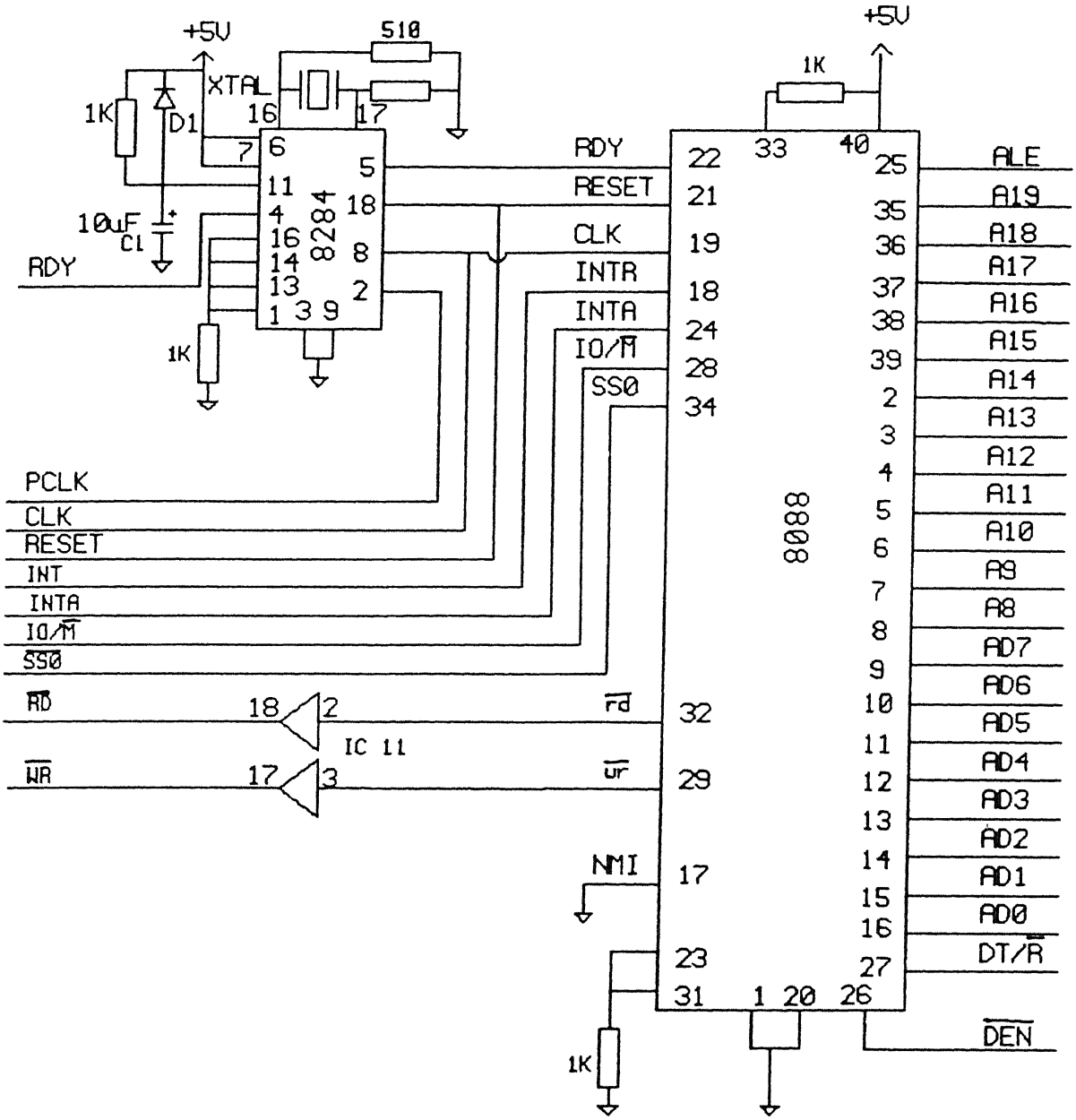


Fig.4.1 CPU and Clock Circuit

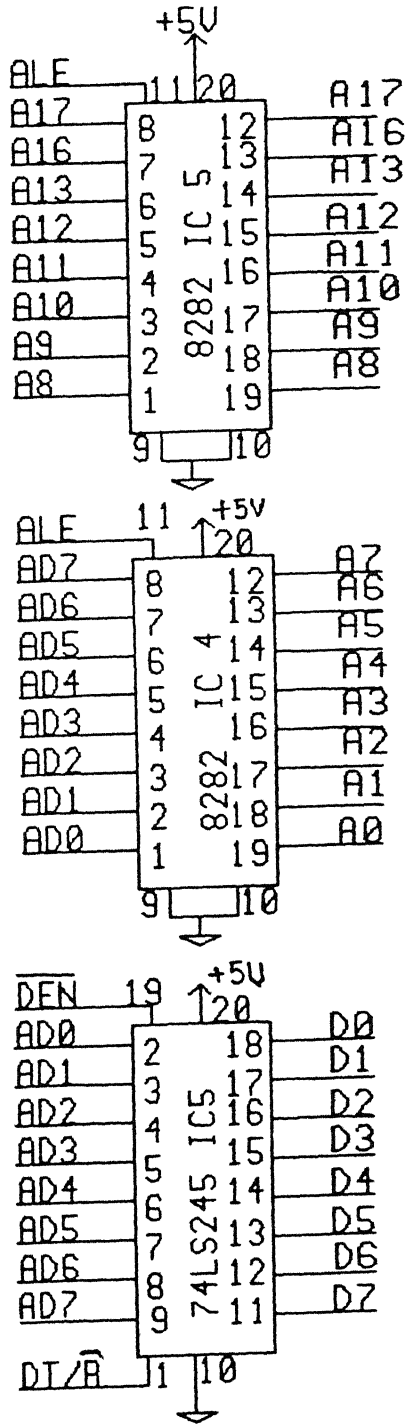


Fig 4.2 Address/data demultiplexing and buffering logic

4.1.1 Address decoding logic

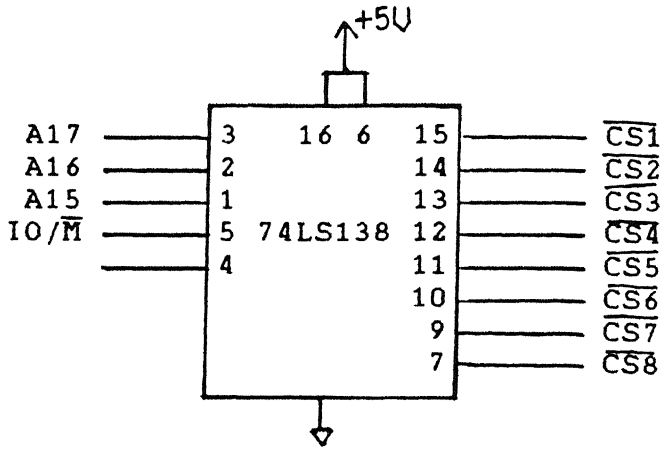
All the addressable devices on the board are memory mapped. A 74LS138 has been used to decode the address lines A15 to A17, to generate Chip-Selects. The decoding logic and the memory map are shown in Fig. 4.3

4.1.2 Memory

The board has one socket for ROM, and two sockets for RAM. Anyone of the ICs 2764, 27128 or 27256 can be used as ROM and any one of the ICs 6264, 62128 or 62256 can be used as RAM. Depending on the type of memory chips used the on board memory, varies from 8K ROM and 8K RAM to 32K ROM and 64K RAM. The jumper connections to be made depending on the memory chips used, are shown in Table.4.1.

4.1.3 Wait state generator

The wait state generator inserts three wait states in read/write cycles to the timer(8253/8254), MPSC-8274. The wait states are also inserted in the interrupt acknowledge(INTA) cycles. The circuit of the wait state generator is shown in Fig. 4.5. The D flip-flops are preset by ALE and $\overline{Q1}$ goes low. $\overline{Q1}$ is high after three low to high clock transitions. $\overline{Q1}$ is gated with chip-selects of the timer, MPSC, and the decoded INTA status. The decoded INTA status is active(low) when IO/$\overline{M}$ =1, $\overline{SS0}$ = 0, and DT/$\overline{R}$ = 0. The output of the gate is fed to the RDY of 8284. The 8284



Address Map

Device	CS	Address range
RAM	CS1	00000-07FFFH
		40000-47FFFH
		80000-87FFFH
		C0000-C7FFFH
PC I/O PORT	CS2	08000-0FFFFH
		48000-4FFFFH
		88000-8FFFFH
		C8000-CFFFFH
PC I/O STATUS PORT	CS3	10000-17FFFH
		50000-57FFFH
		90000-97FFFH
		D0000-D7FFFH
8253/8254	CS4	18000-1FFFFH
		58000-5FFFFH
		98000-9FFFFH
		D8000-DFFFFH
8274	CS6	28000-2FFFFH
		68000-6FFFFH
		A8000-AFFFFH
		E8000-EFFFFH
RAM	CS7	30000-37FFFH
		70000-77FFFH
		B0000-B7FFFH
		F0000-F7FFFH
ROM	CS8	38000-3FFFFH
		78000-7FFFFH
		B8000-BFFFFH
		F8000-FFFFFH

Fig. 4.3 Address decoding logic

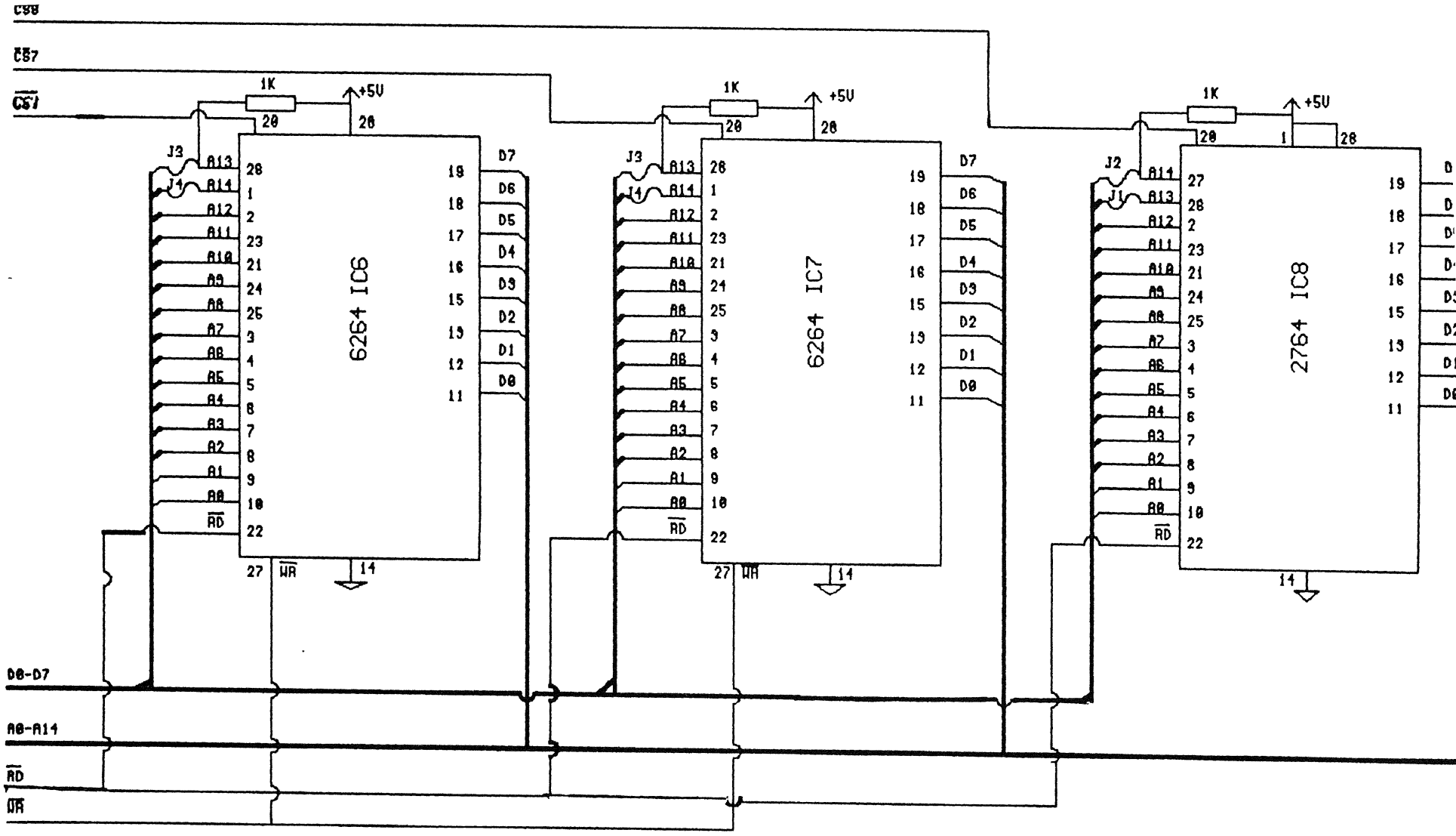


Fig.4.4 Memory circuit

Table 4.1

Jumper Connections

Memory IC	J1	J2	J3	J4
6264	X	X	Open	Open
62128	X	X	Short	Open
62256	X	X	Short	Short
2764	Open	Open	X	X
27128	Open	Short	X	X
27256	Short	Short	X	X

synchronizes the RDY signal, and provides a synchronous READY signal to the CPU. The step-wise functioning of wait state generator is explained in table 4.2

4.1.4 Timer and CPU interface

An 8253-5, timer, is used on the board. However, when the on board system clock is above 5 Mhz an 8254 is used, as timer, to ensure timing compatibility during read/write cycles to the timer. The CPU and Timer interface is shown in Fig. 4.6.

The input clock of Counter-0 is obtained by dividing the PCLK, of 8284, by two. The PCLK is half the system clock. The division of PCLK ensures that the clock input to the timer is less than 2 Mhz (the max. limit for 8253).

4.1.5 MPSC-8274

The 8274 Multi-Protocol serial controller, is capable of handling both asynchronous and synchronous communication protocols. It's programmable features allow it to be configured in various modes. It has two independent serial I/O channels, Channel A and Channel B. Each channel can be configured into full duplex mode and may operate in a mode or protocol different from the other channel, possibly at differing speeds.

On the serial I/O side the transmit section, of each channel of 8274 consists a serial shift register. The

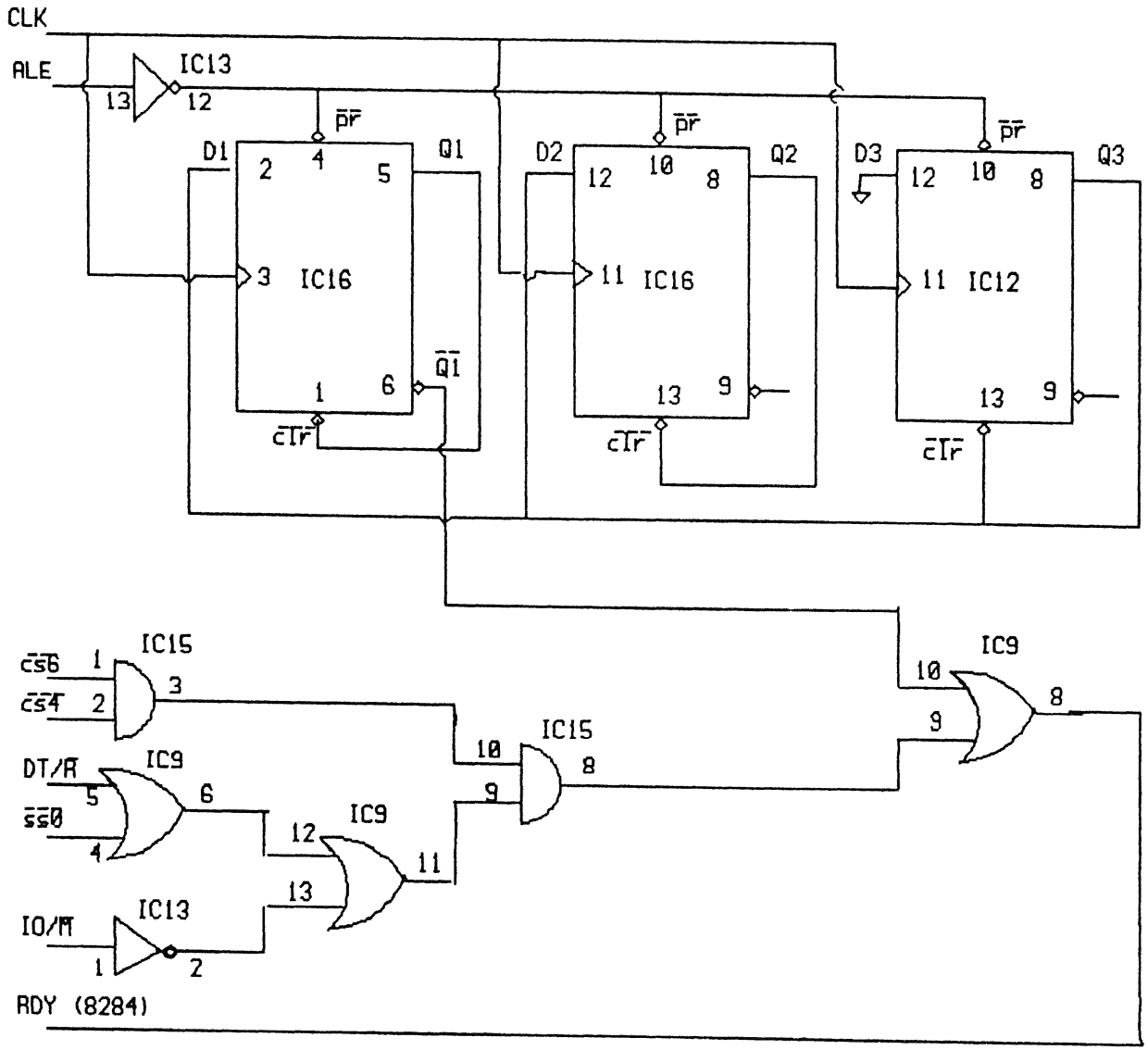
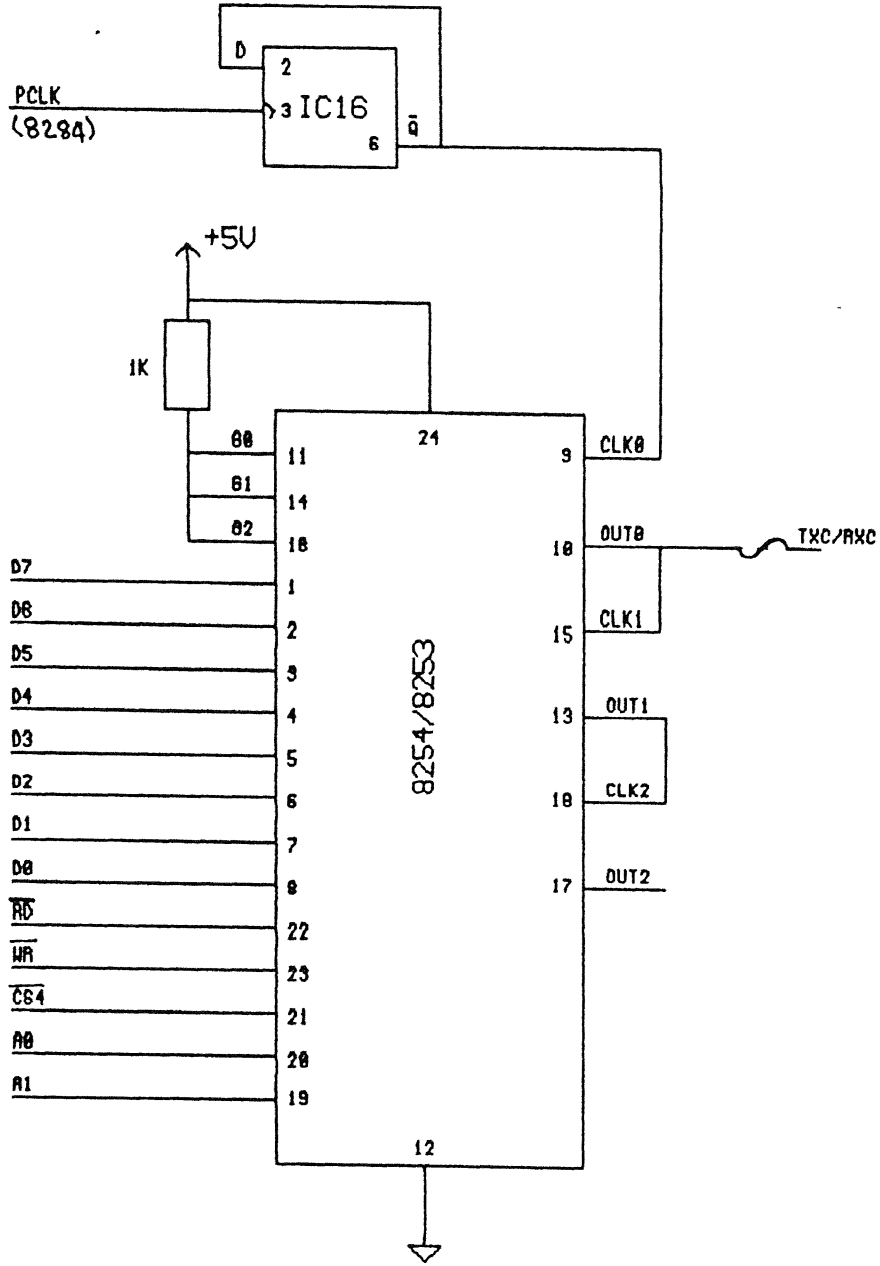


Fig.4.5 Wait state generator

Table 4.2

Stepwise functioning of Wait state generator

State	$\overline{cs}$ = $\overline{cs3}$ + $\overline{cs5}$ + decoded INTA status	$\overline{Q1}$	$\overline{Q2}$	$\overline{Q3}$	RDY	Remarks
T1	1	0	0	0	1	
T2	0	0	0	0	0	
T3	0	0	0	1	0	
TW1	0	0	1	1	0	Three wait
TW2	0	1	1	1	1	states are
TW3	0	1	1	1	1	inserted
T4	1	1	1	1	1	
T1	1	0	0	0	1	No wait
T2	1	0	0	0	1	states
T3	1	0	0	1	1	are
T4	1	0	1	1	1	inserted
T1	1	0	0	0	1	


D
2
PCLK
(8284)
3 IC16
6
Q̄
+5V
1K
24
A0
11
A1
14
A2
18
D7
1
D6
2
D5
3
D4
4
D3
5
D2
6
D1
7
D0
8
RD
22
WR
23
CS4
21
A0
20
A1
19
8254/8253
12
9
CLK0
10
OUT0
TXC/RXC
15
CLK1
13
OUT1
18
CLK2
17
OUT2

transmit data in serial shift register is shifted out through a two bit delay onto the TX data line. The two bit delay is used to synchronize the internal shift clock to the transmit clock. In synchronous communication mode the data in shift register is also presented to zero bit insertion logic which inserts a zero after sensing five contiguous ones in data stream. In this mode the CRC generator is in parallel, computing the CRC on transmitted data and appends the frame transmission with the CRC bytes at the end of data transmission.

In the receive section, the received data is transferred to three byte deep FIFO. The data is transferred to the top of FIFO at the chip clock rate. In SDLC/HDLC mode the incoming data is continuously monitored for flags and inserted zero bits, and they are deleted from the data presented to the CPU interface. The receive section performs the CRC check, also.

On the board the features of MPSC are exploited to a great extent. The MPSC interface is shown in Fig. 4.6. The interrupt line of the MPSC and that of the interrupt-generator are connected in daisy-chain configuration. The interrupt priority pin (IPI) is grounded, to provide highest priority to the interrupts raised by the MPSC. The interrupt priority output pin (IPO) is ORed with the INTA signal, and the output of the OR gate is given to the enable of the vector-generator (IC 22). The vector generator is enabled

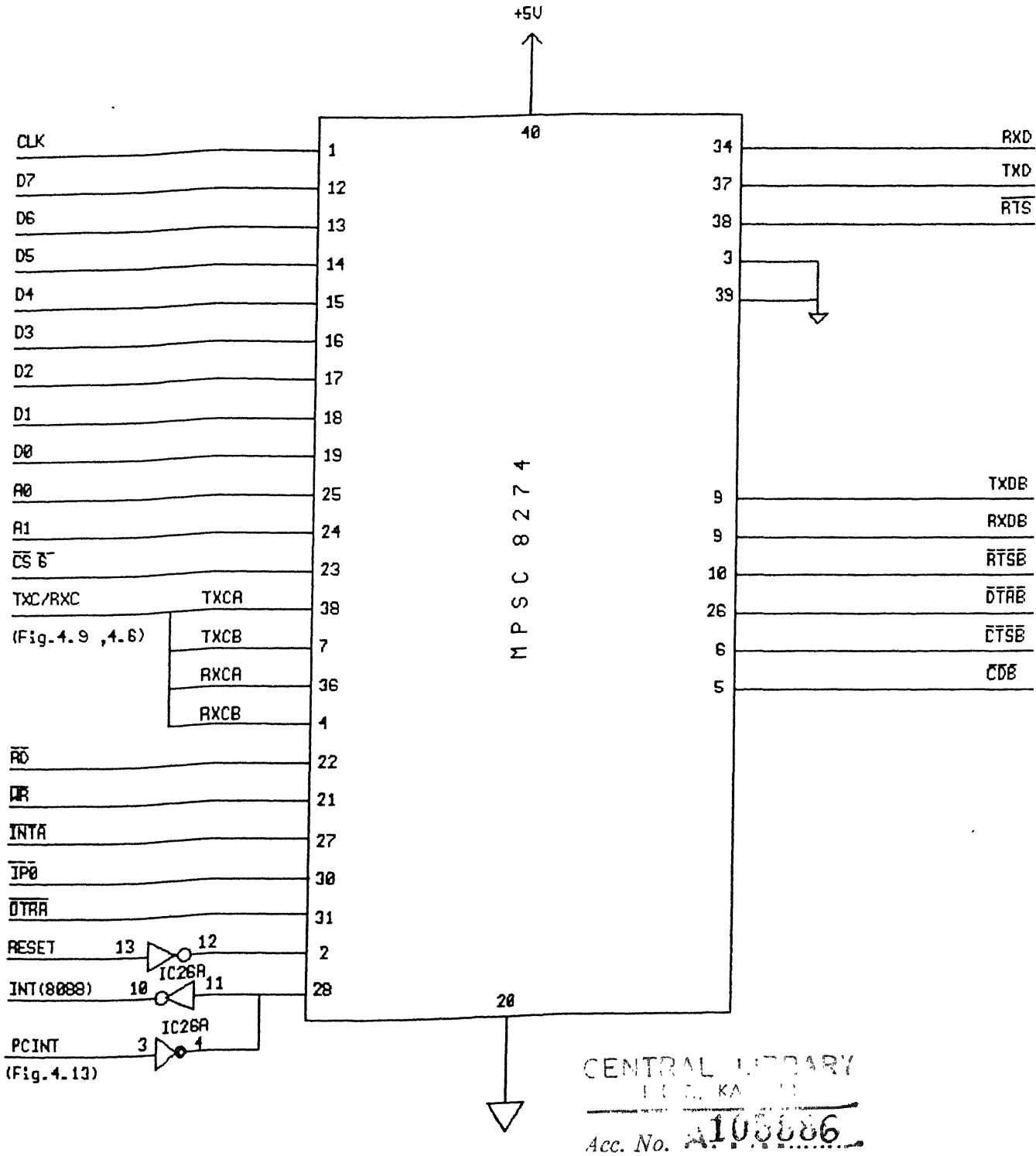


Fig.4.7 MPSC interface

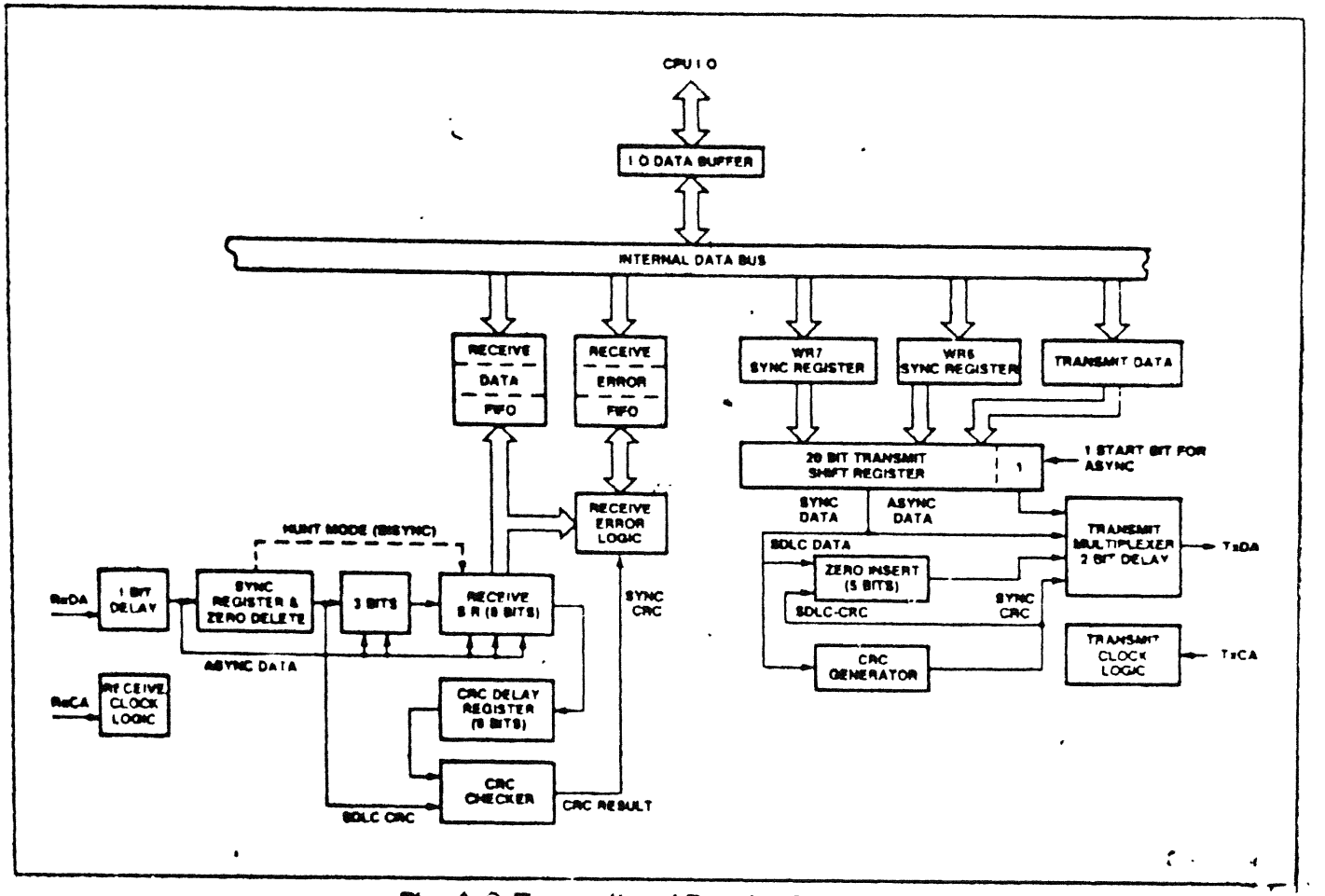
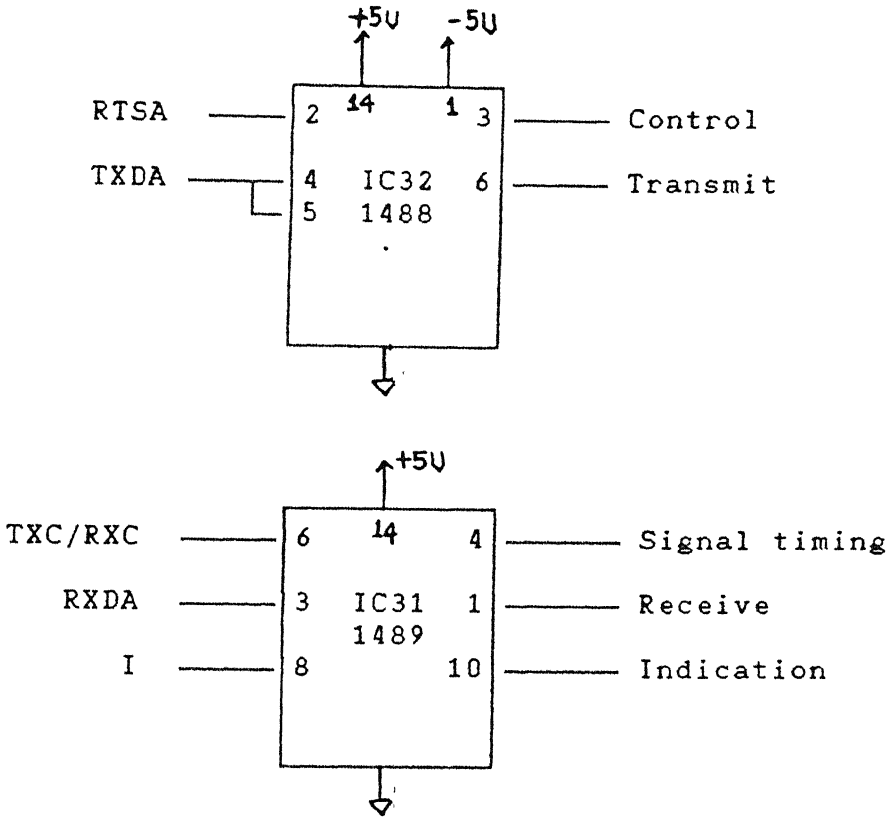
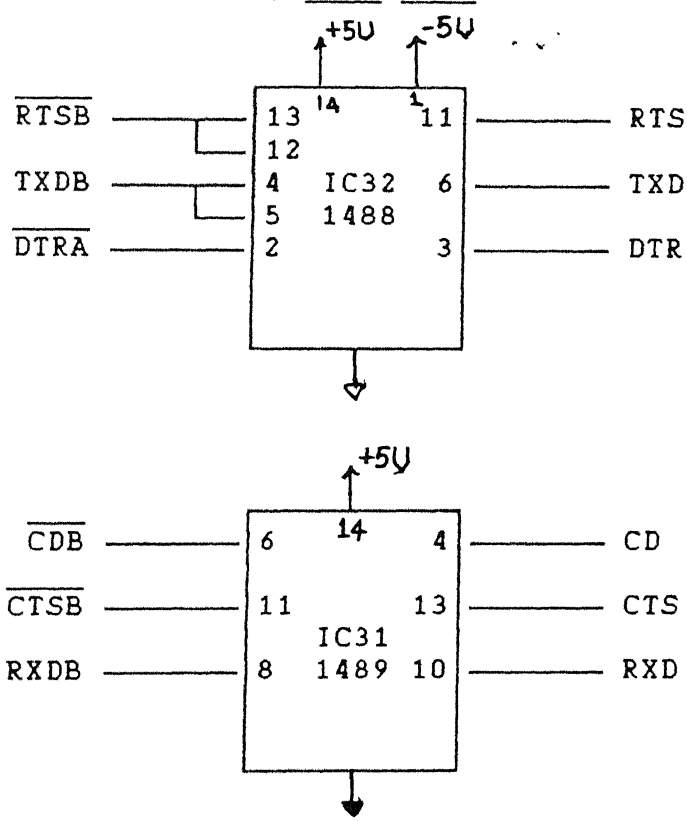


Fig 4-8 Transmit and Receive Data Path



a) X.21 Port



b) RS-232 Port

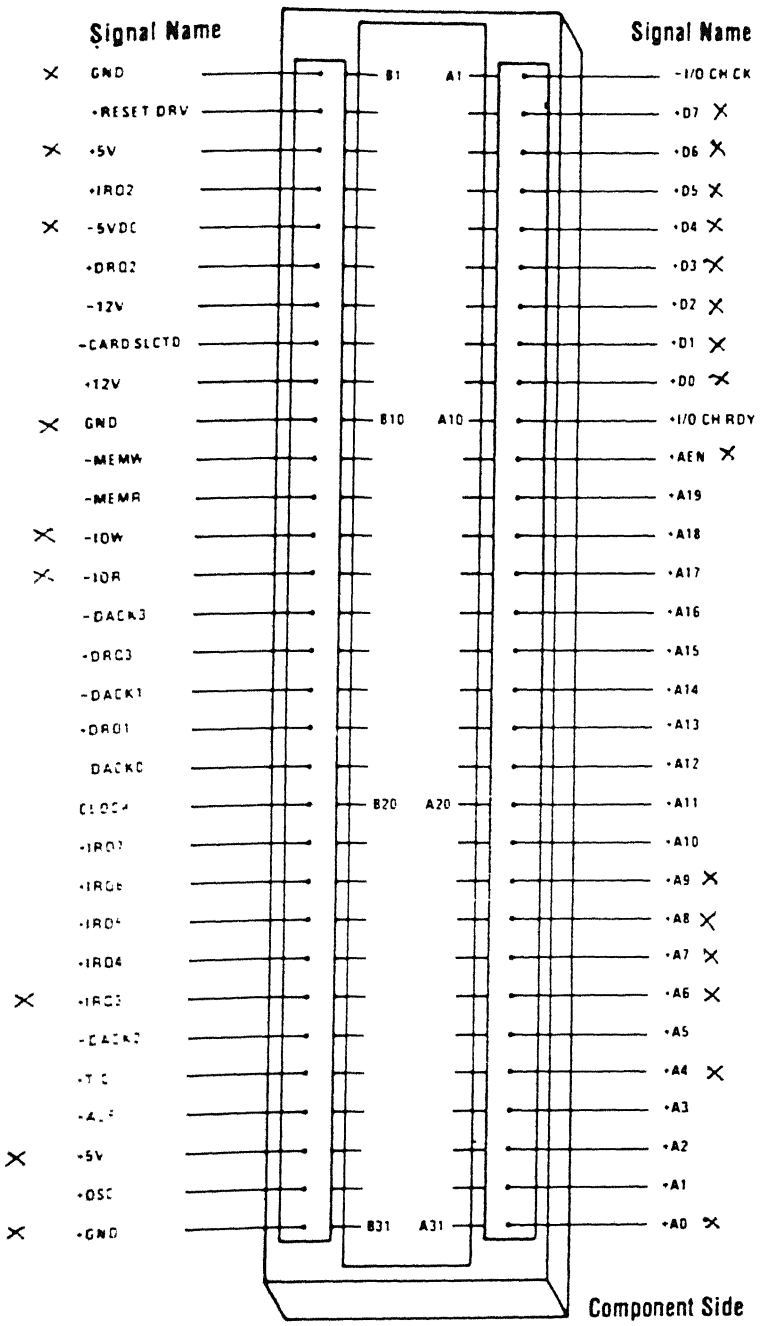
4.9 Line Drivers/Receivers

when both INTA and IPO are active (low). When MPSC raises the interrupt, the IPO pin is inactive(high), disabling the vector generator, and the MPSC places the vector on the data bus during the INTA cycle. When the interrupt line is activated by the interrupt-generator only, the IPO pin is active(low) and the vector generator places the vector 'FF' on the data bus during the INTA cycle.

The channel A of MPSC is used to support the X.21 interface. The DTR, of channel A, is used as control ('C') signal. The incoming indication ('I') signal is read by the processor through the PC I/O status port. The transmit and receive clock, i.e. the signal element timing ('S'), is fed externally. However, for testing in loop back mode the receive and transmit clock can be derived from the Counter-0 (8253/8254). Channel B, also, can receive it's transmit and receive clock either from Counter-0 or external world. The channel B communication protocol is not defined, but it can be programmed in synchronous or asynchronous mode. The serial I/O signals are level translated (TTL/RS232-C) using the 1488 drivers(IC29, IC32) and 1489 receivers(IC30, IC31).

4.2 PC INTERFACE

The board is interfaced to the PC, through the PC-Bus. The PC-BUS signals are shown in fig.4.10, and the signals, of it, used on board are marked in the fig.. The data transfer between the PC and the board is through the PC I/O



I/O Channel Diagram

Fig. 4.10 PC Bus

Table 4.3

PC I/O Address Space

Hex Range	Usage
000-00F	DMA Chip 8237A-5
020-021	Interrupt 8259A
040-043	Timer 8253-5
060-063	PPI 8255A-5
080-083	DMA Page Registers
0AX*	NMI Mask Register
0CX	Reserved
0EX	Reserved
200-20F	Game Control
210-217	Expansion Unit
220-24F	Reserved
278-27F	Reserved
2F0-2F7	Reserved
2F8-2FF	Asynchronous Communications (Secondary)
300-31F	Prototype Card
320-32F	Fixed Disk
378-37F	Parallel Printer
380-38F	SDLC Communications
3A0-3AF	Reserved
3B0-3BF	IBM Monochrome Display/Printer
3C0-3CF	Reserved
3D0-3DF	Color/Graphics
3E0-3E7	Reserved
3F0-3F7	Diskette
3F8-3FF	Asynchronous Communications (Primary)

port on board. The board uses the interrupt line IRQ3 of PC, as explained in 4.2.3. Since the IRQ3 line is also used by COM2 port, of the PC, to avoid contention the COM2 interrupts are to be disabled through appropriate jumper connections on the COM2 adapter.

4.2.1 Address decoding logic

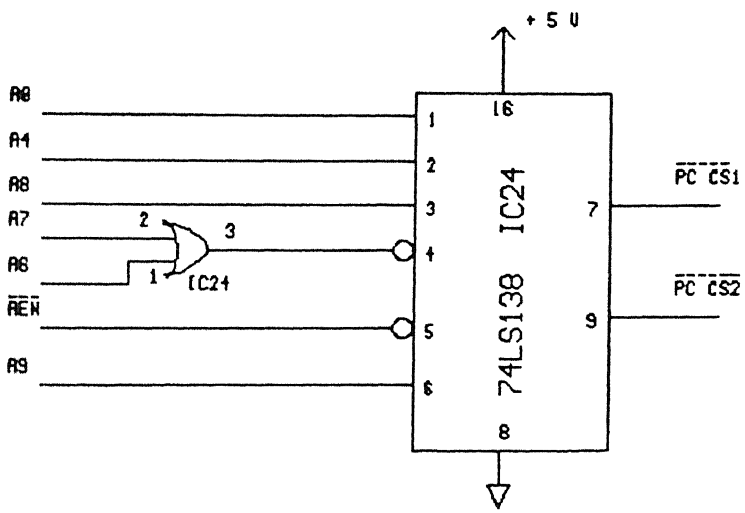
The IBM PC does not use the full I/O address space of 8088. It uses only the lower 10 bits of the address. The peripheral card I/O address space requires the bit 9 to be zero. The complete I/O address space of the PC is shown in table 4.3. The address lines of the PC are decoded, using 74LS138, to provide chip-selects to the PC I/O status port and the PC I/O port. The decoding logic and address map are shown in fig.4.11.

4.2.1 PC I/O Port

The PC I/O port is realized using two 8282s, as shown in fig.4.12. For data transfer between the board and the PC, the board processor writes to IC27 and reads from IC28 while the PC processor writes to IC28 and reads from IC27.

4.2.2 PC I/O Status port

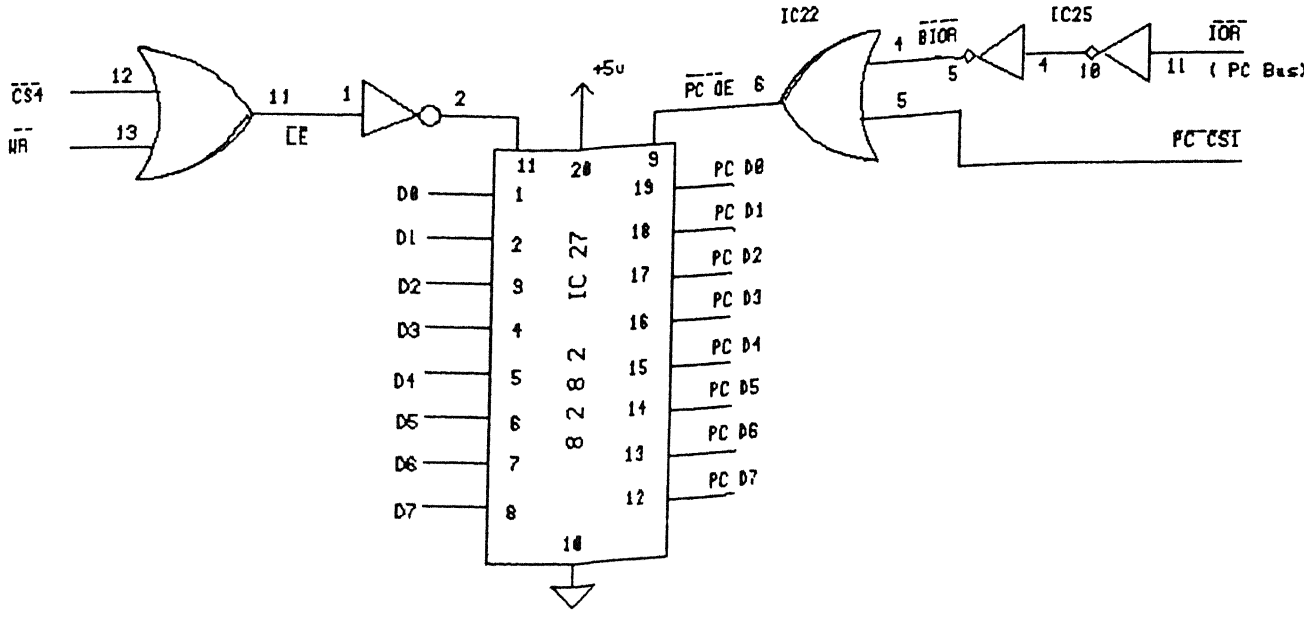
The PC Status I/O port implementation is shown in fig. 4.13. On reading this the board processor obtains the status of I line of X.21 port, PC input port full/empty, and PC output port full/empty. Reading this port PC processor



ADDRESS MAP

$\overline{CS}$	Address range
$\overline{PCCS1}$	All odd addresses in 310-33F H 330-33F H
$\overline{PCCS2}$	All even addresses in 310-31F H 330-33F H

Fig. 4.11 PC I/O ADDRESS DECODING LOGIC



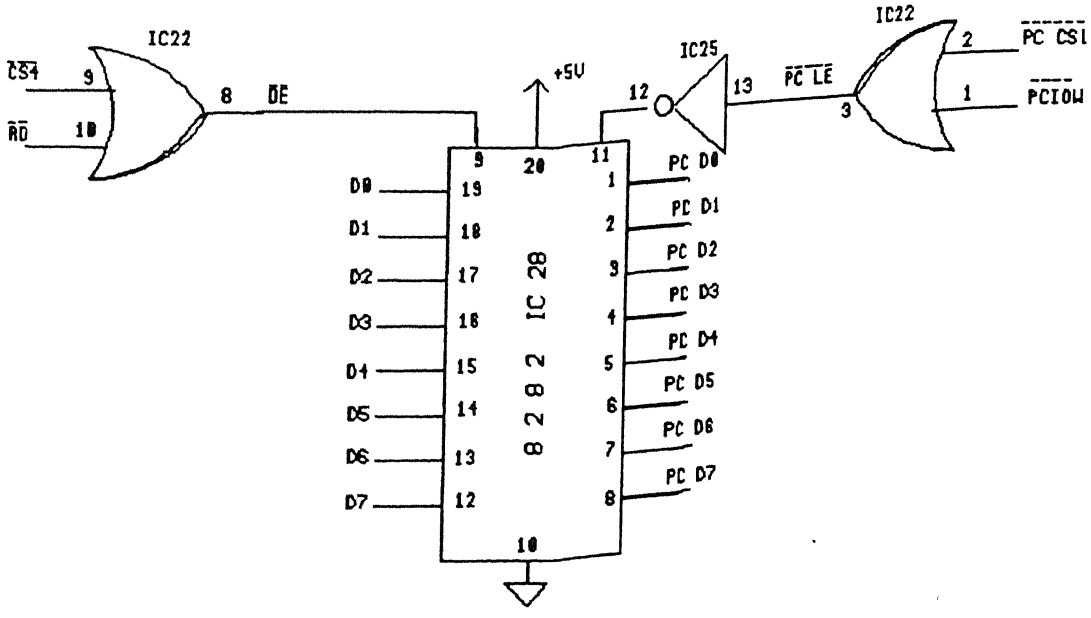


Fig.4.12 PC I/O Port

can obtain the status of PC I/O write port full/empty, and card busy/not busy status.

4.2.3 Interrupt generator

The interrupt generator logic is shown in fig. 4.14. Whenever PC writes a byte to the PC I/O port, the flip-flop (IC19) is set and the PCINT goes low interrupting the on board processor. During the INTA cycle of this interrupt the vector generator places the vector 'FF' on the data bus. When the on board processor reads the PC I/O port, the above said flip-flop is cleared and the PCINT goes high. Similarly when the on board processor writes a byte to the PC I/O port, the IRQ3 line of PC is activated and is deactivated when the PC processor reads the PC I/O port.

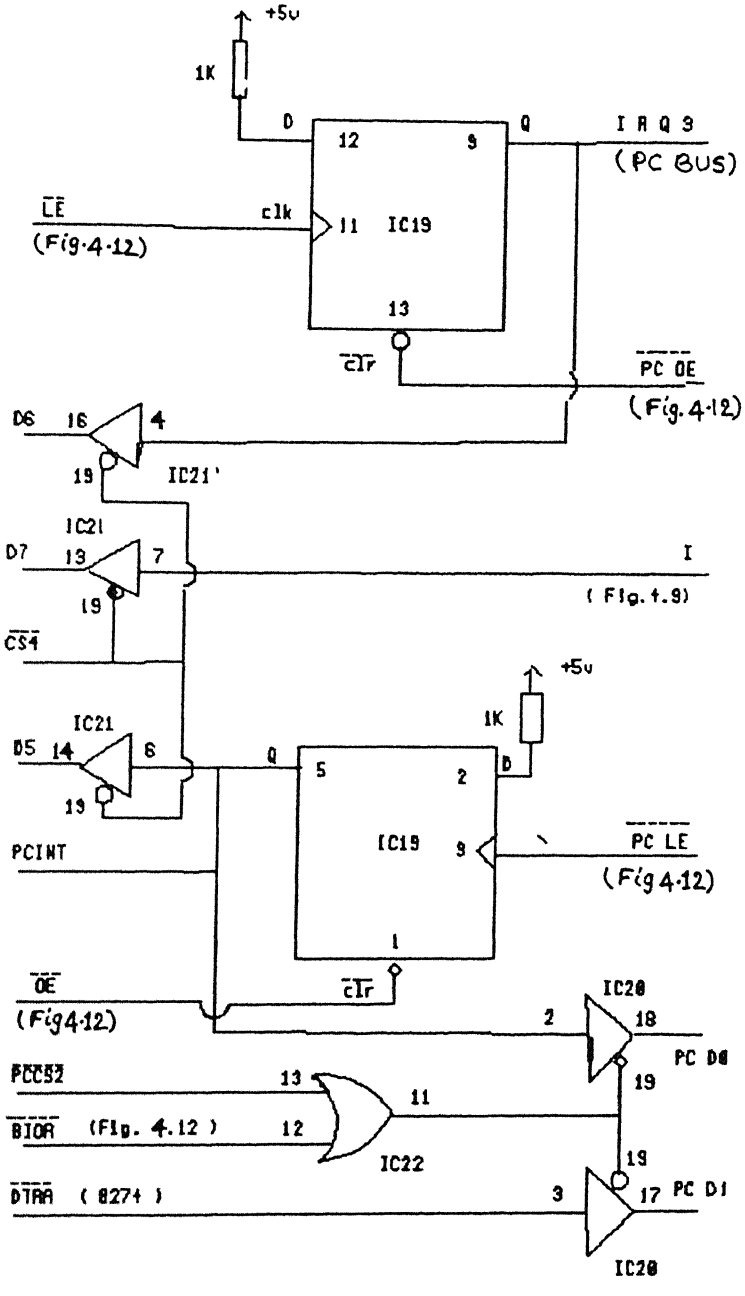


Fig.4.13 PC I/O port
and Intrrupt generator

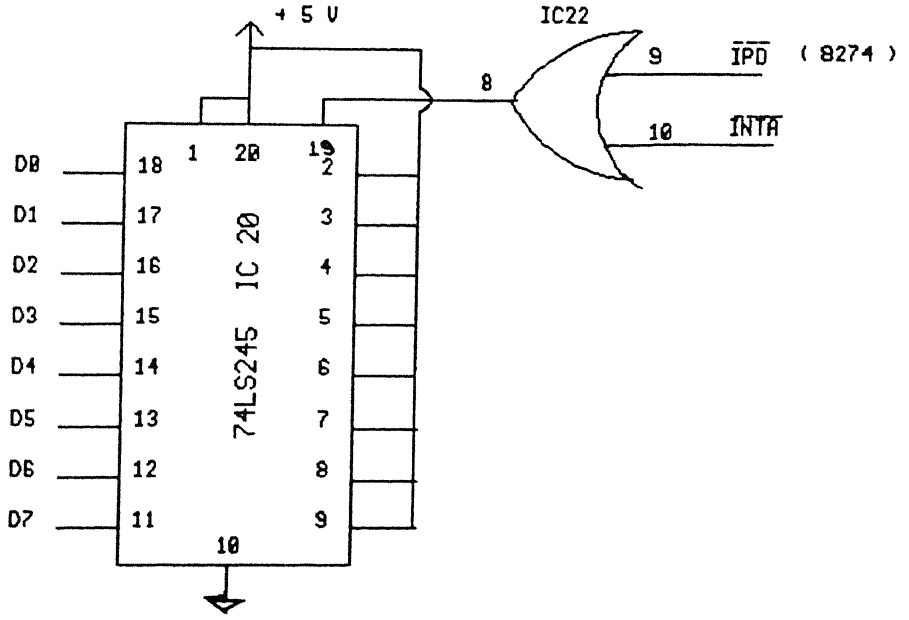


Fig.4.14 Vector generator

CHAPTER 5

FIRMWARE IMPLEMENTATION

The general aspects of the firmware are explained in chapter 3. Here the firmware implementation and the different routines of the firmware are discussed in detail with the help of flow charts. As already explained in chapter 3, the firmware is classified into :

1. Initialization Software
2. Data transfer Software
3. Interrupt service routines

5.1 INITIALIZATION SOFTWARE

On 'Power on reset ', the Processor program execution begins from the address OFFFF0 H. From this address, using a far JUMP instruction, control is transferred to the initialization routine. The flow-chart of this routine is shown in Fig.5.1. The functions of this routine are, as given below :

1. Initialize Segment registers of CPU, Stack and Vector table
2. Program 8253/8254 timers in mode3, with counter2 programmed to count from 00 to OFFFFH at the rate of one count per 0.1 sec.

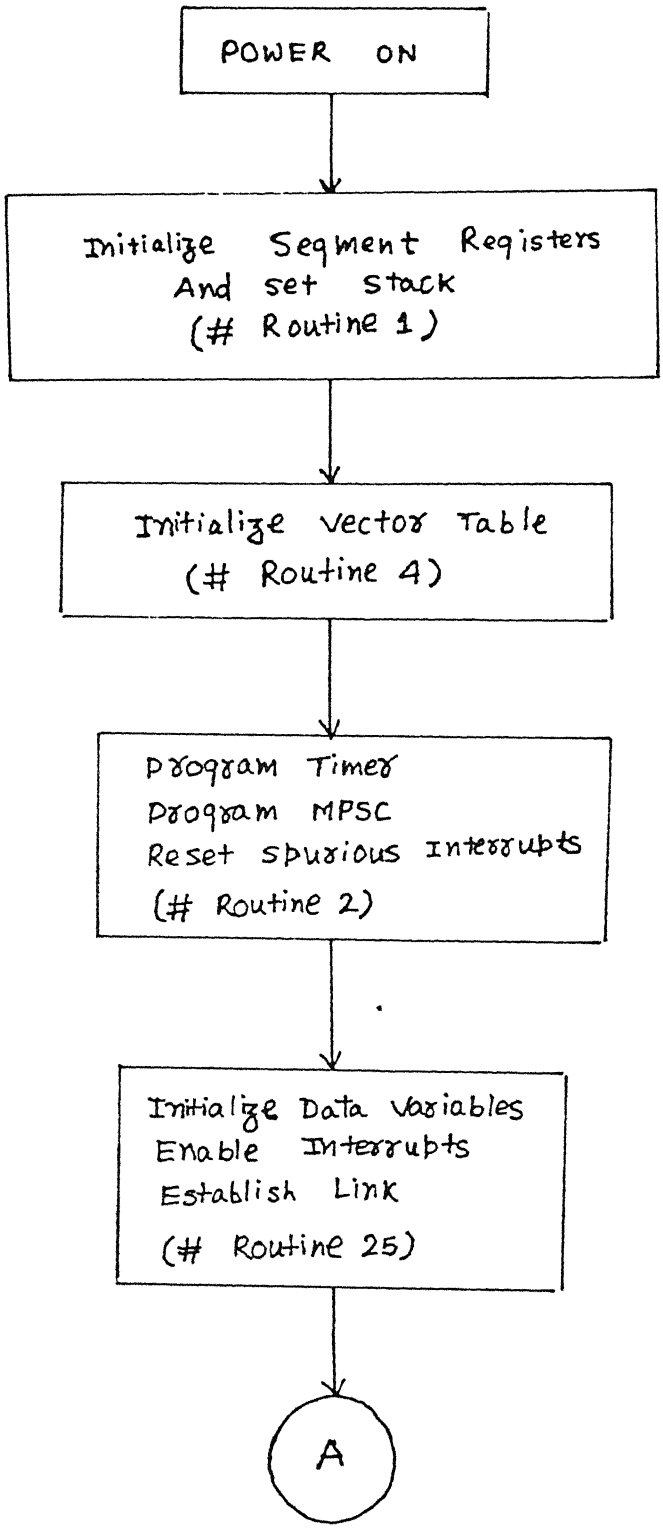


Fig 5.1 Flow-chart of Initialization Software

3. Reset both the channels of MPSC-8274 [7], and program channel A in SDLC/HDLC mode with CCITT-CRC enabled, address search mode disabled, and in 8086/8088 vectored interrupt mode. All the interrupts of channel A are enabled and those of channel B are disabled. The relative priorities among the internal MPSC interrupt sources is programmed to be in the following order :
   Rx-A, Tx-A, Rx-B,Tx-B,Ext-A,Ext-B [7].
   The error reset and the reset external status commands are issued to both the channels of MPSC to reset spurious interrupts raised by MPSC [7]. The PC I/O port is read to reset the spurious interrupts raised by this port.
4. Initialize the various data variables, in data segment, establish the physical link and enable the interrupts.
5. Transmit 'SABM' command, to establish the balanced, data link. The link is established on receiving an 'UA' response.

After performing these functions, control is transferred to the 'Data transfer' routine.

5.2 DATA TRANSFER SOFTWARE

The flow-chart of this software is shown in Fig.5.2 to Fig.5.8. This routine is in a infinite loop, and the various

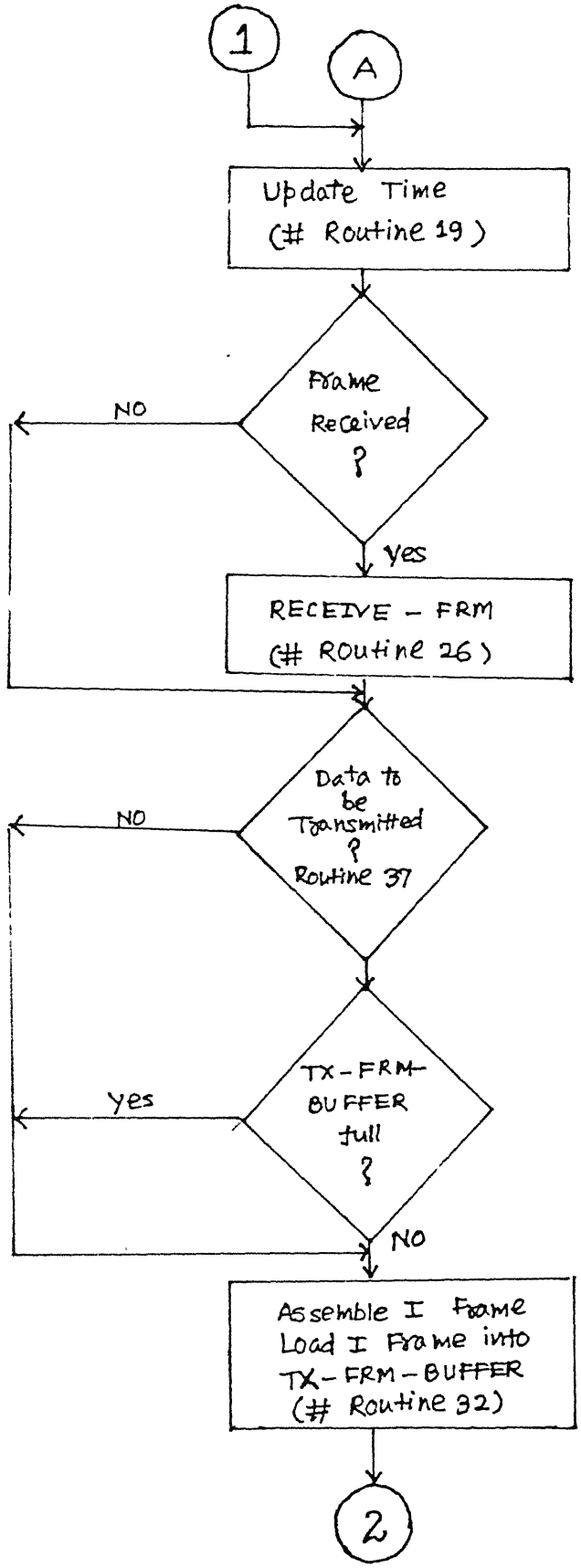


Fig 5.2 Flow chart of Data Transfer Software (contd...)
(# Routine 25)

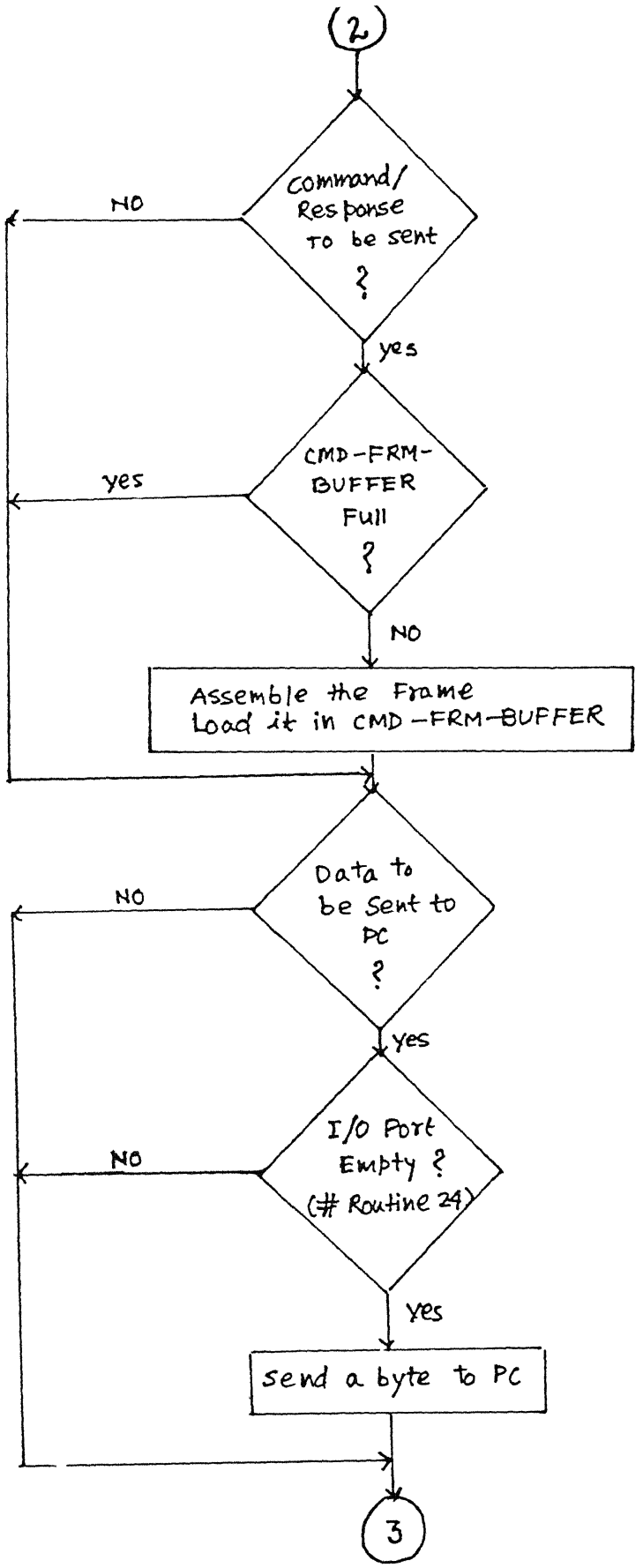


Fig 5.3 Flow chart of DATA_TRANSFER software (contd···)
(# Routine 25)

functions of this routine are as given below :

1. Read 8253/8254 and update the time.
2. Compare the base pointers of the receive frame buffer (RX_FRM_BUFFER), to determine if a frame is received and, call 'RECEIVE_FRM' routine, to analyze the received frame and to take appropriate action.
3. Compare the base pointers of transmit packet buffer (TX_PKT_BUFFER) to determine if there is data to be transmitted. Assemble I frames of this data, and load them into the transmit window (TX_FRM_BUFFER), provided the number of frames in this window are less than seven. A frame in this window is discarded, releasing buffer space, only after it is acknowledged.
4. Load RR or RNR frames into CMD_FRM_BUFFER, depending on the local ready state, under the following conditions :

   i) A received I frame is not acknowledged with in a period 'SEND_ACK_TIME_OUT', after it's receipt.

   ii) There is a change in local ready state.

   The RR or RNR frames are also transmitted, with F bit set, when a frame with P bit set is received.
5. Reset send state variable (V_S) to receive

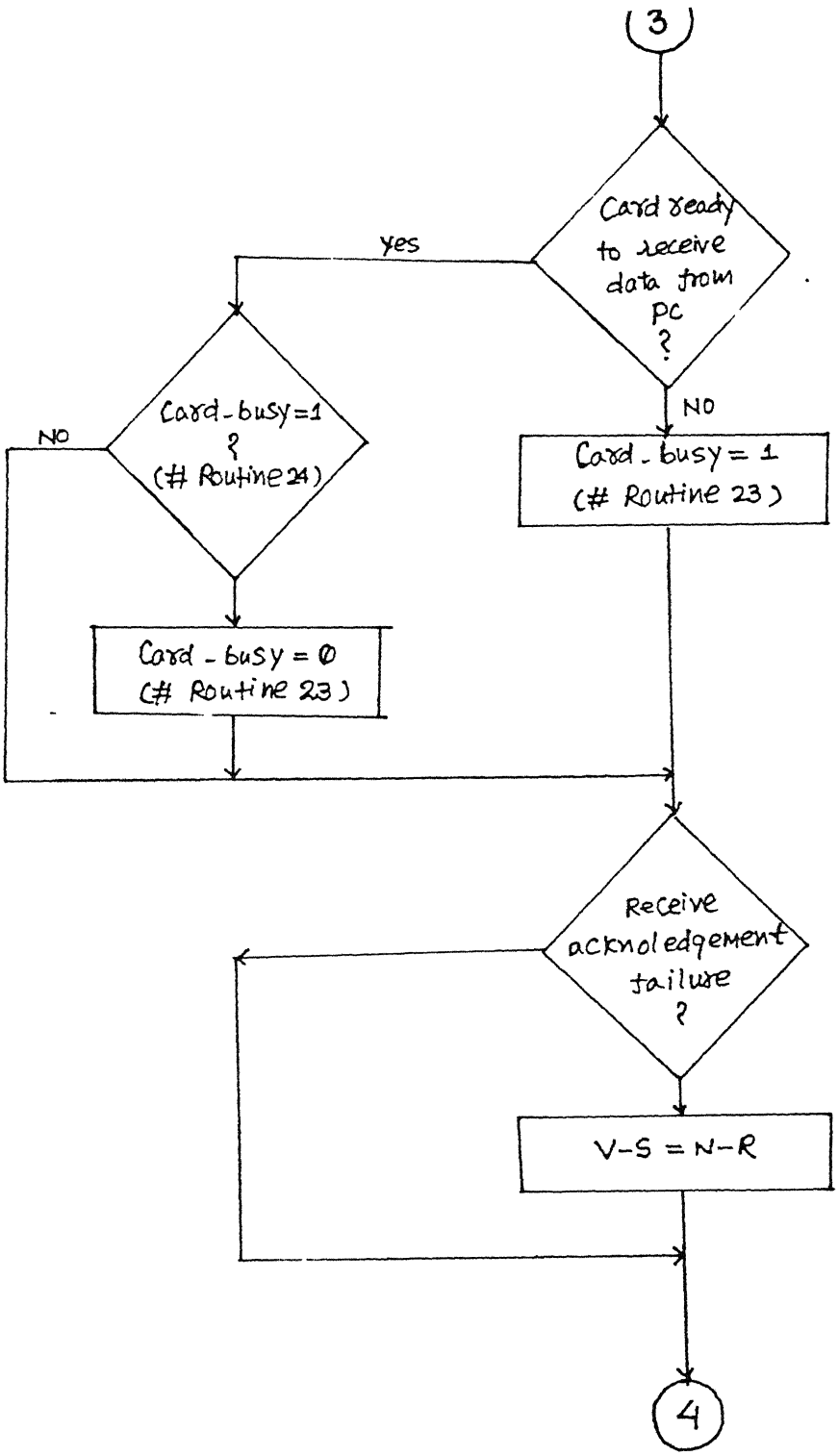


Fig 5.4 Flow chart of DATA-TRANSFER Software (..Contd..)

sequence variable (N_R) and re-transmit all the unacknowledged frames, in transmit window (TX_FRM_BUFFER), if anyone of the receive acknowledgment timer expires.

6. Load the first character of the frame to be transmitted into the transmit frame buffer to enable transmit buffer empty interrupt, if this interrupt is in reset state and the destination is ready.
7. Send the data in receive packet buffer (RX_PKT_BUFFER) to PC, through PC I/O port. The data is written to PC I/O port only if the port is empty.

The 'RECEIVE_FRM' routine's flow-chart is shown in Fig.5.6 to fig.5.8. This routine analyzes the control field of received frame and depending on the type of frame received, the actions taken are as given below :

1. UN Frame : If the frame is a 'SABM', the send and receive state variables are reset to zero, and an 'UA' response frame is transmitted, to re-establish the link. On receiving a 'DISC' or 'DM' frame the link is disconnected, and remains disconnected till the link is re-established.
2. NON - UN FRAME: The received frame's acknowledgment field (i.e.,N_R) is read and is accepted, only if it's value is greater than the previous

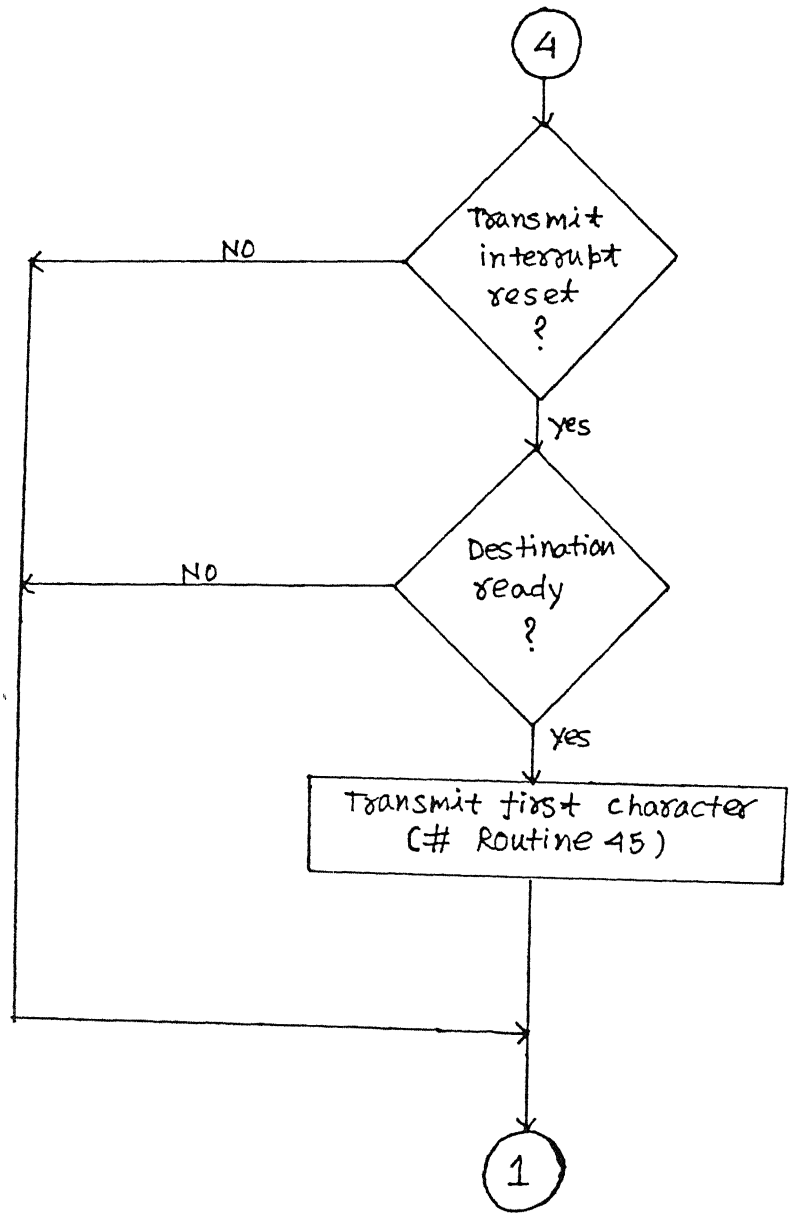


Fig 5.5 Flow Chart of DATA-TRANSFER software (.. Contd.)

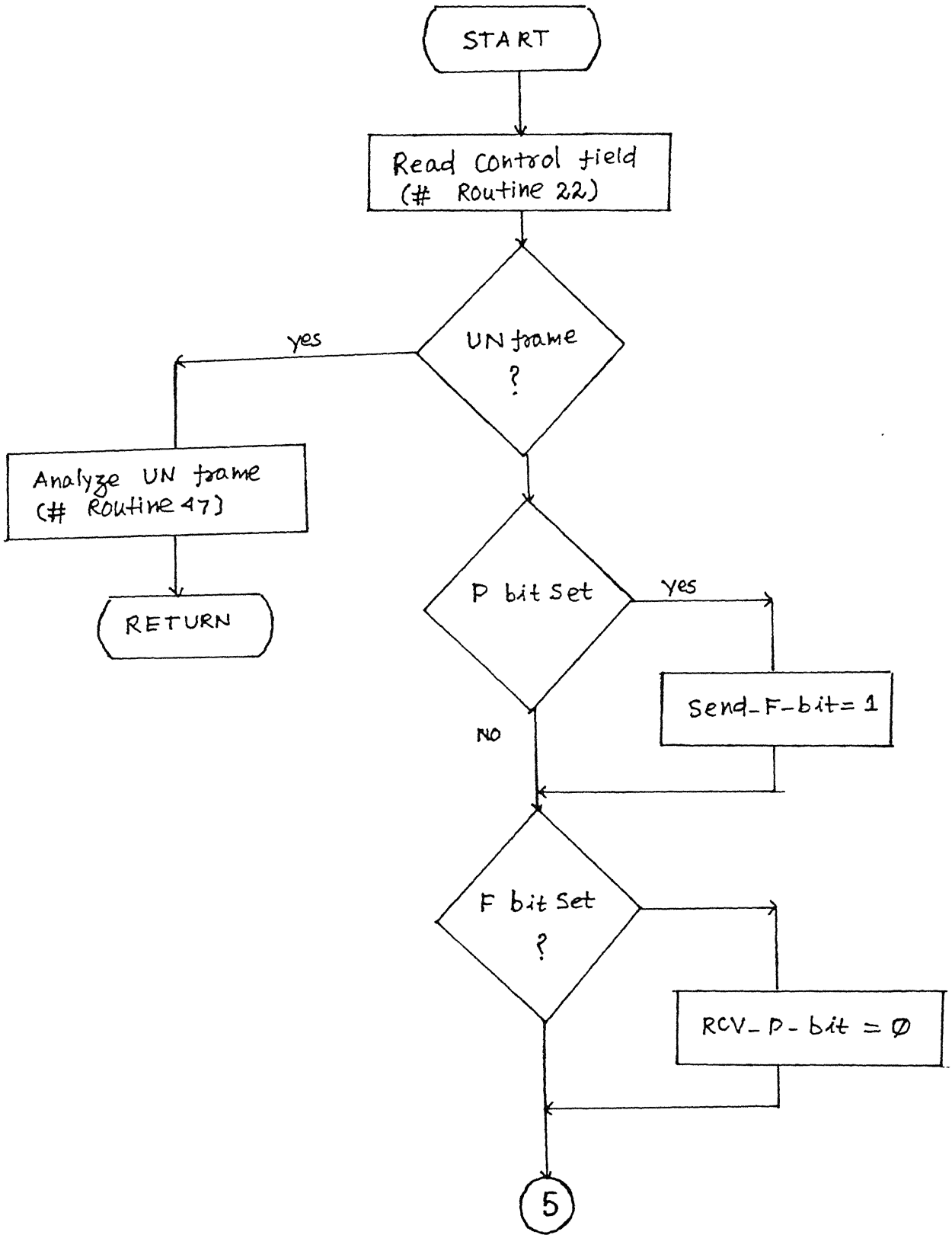


Fig 5.6 Flow Chart of RECEIVE-FRM Routine (Contd..)
(# Routine 26)

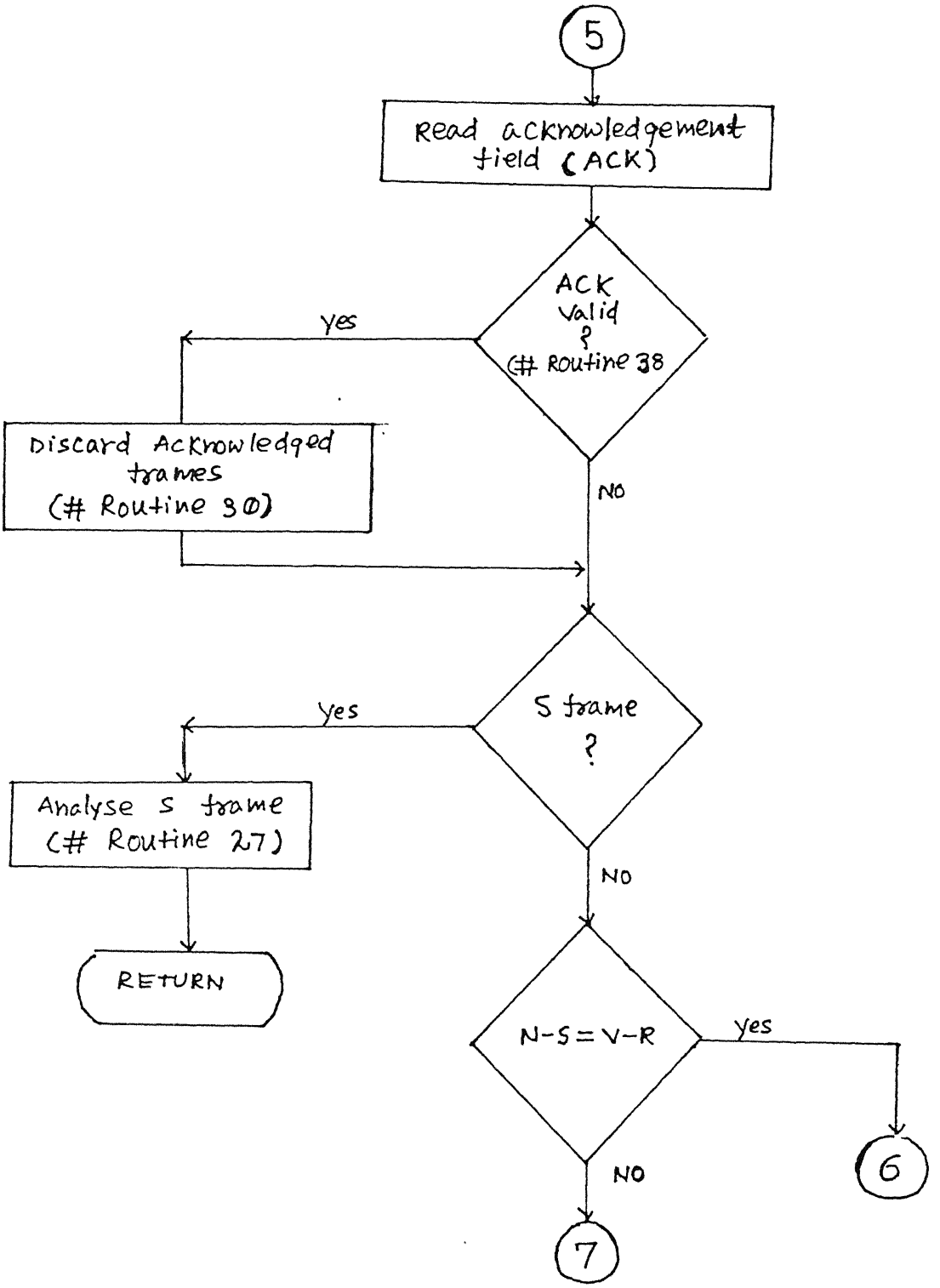


Fig 5.7 Flow Chart of RECEIVE_FRM Routine (Contd..)
(# Routine 26)

acknowledgment and less than the send sequence number N_S. On accepting the acknowledgment, the acknowledged frames are discarded from the transmit window (TX_FRM_BUFFER), and reject status (REJECT_STAT) is reset to zero.

On receiving a RR or RNR frame, the status of destination is updated. A received reject frame (REJ) is accepted only if the reject status is zero and it's acknowledgment is valid. On receiving a valid reject frame the send state number (V_S) is reset to receive sequence number (N_R) and all the unacknowledged frames are re-transmitted.

A received I frame is accepted only if it's send sequence number (N_S) matches the local receive state number(V_R). In-case of mismatch a REJ frame is loaded into the CMD_FRM_BUFFER. The information content of the accepted I frame is transferred to Receive packet buffer (RX_PKT_BUFFER).

After analyzing the received frame it is discarded from RX_FRM_BUFFER. However, a valid frame is not discarded from the buffer if the information content of the frame is not read or an appropriate response frame is not transmitted, due to buffer space limitations in other buffers.

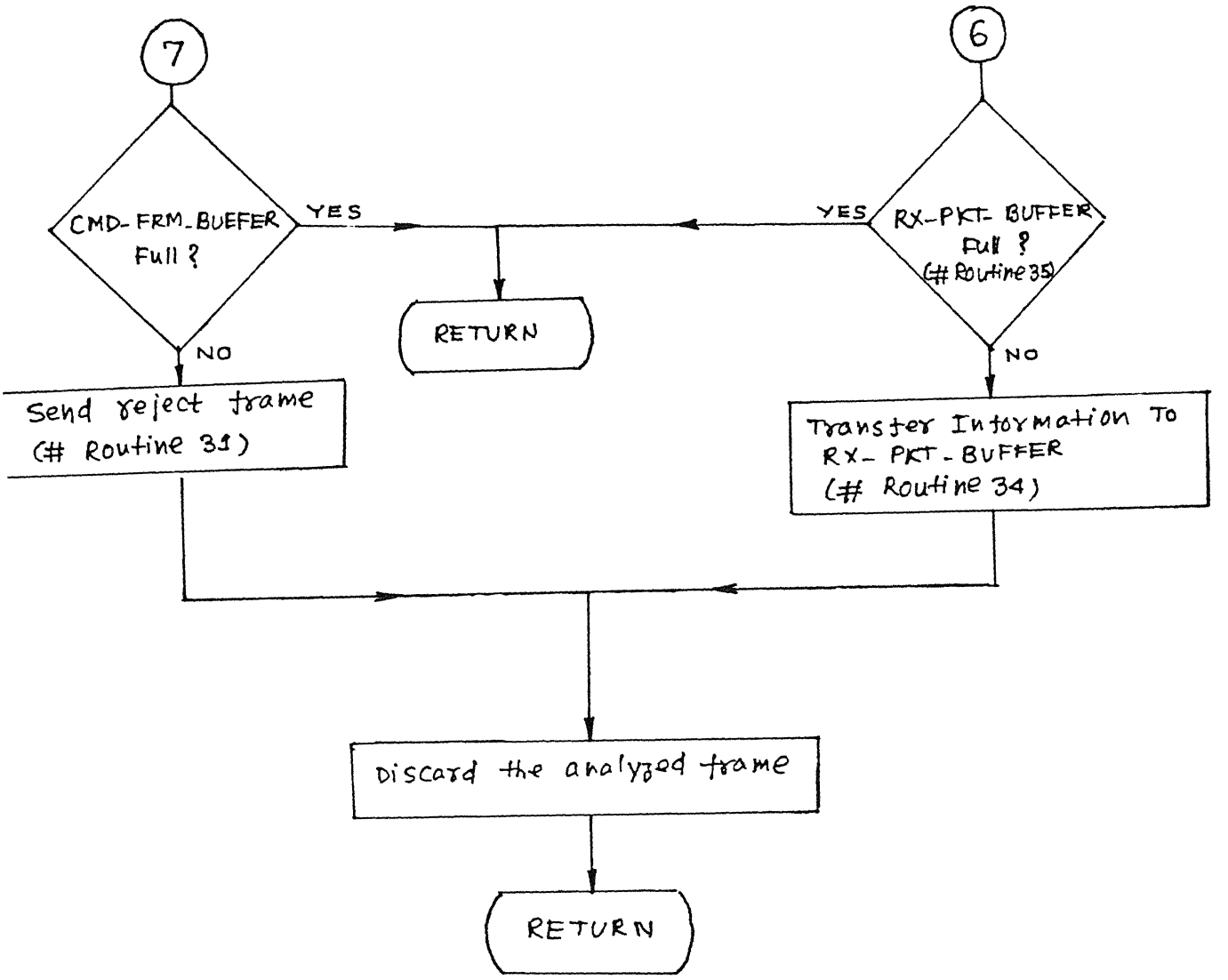


Fig 5.8 Flow chart of RECEIVE-FRM Routine.
(# Routine 26)

5.3 INTERRUPT SERVICE ROUTINES (ISRs)

The interrupts on card are raised by the MPSC-8274 [7] and PC I/O port. The various sources of interrupts and the functions performed in their interrupt service routines are given below :

1. Transmit buffer empty : This invokes the 'TX_ISR', whose flow-chart is shown in Fig.5.9. In the ISR the next byte of frame under transmission is loaded into the transmit-buffer of MPSC-8274. After loading each byte the end of message latch, of 8274, is reset, to ensure the transmission of CRC bytes at the end of frame transmission. If the frame under transmission is an I frame, the data to be transmitted is read from the transmit window (TX_FRM_BUFFER), otherwise it is read from CMD_FRM_BUFFER. When there are no more bytes to be transmitted, the transmit buffer empty interrupt is reset. This interrupt reset results in transmitter underrun, and the CRC bytes followed by the closing flag are transmitted. After transmitting the closing flag the transmit buffer empty interrupt re-occurs and it is reset.

2. Transmitter underrun/End of Message : This invokes the 'EOM' ISR, whose flow-chart is shown in Fig.5.10. If the frame transmitted is an I frame,

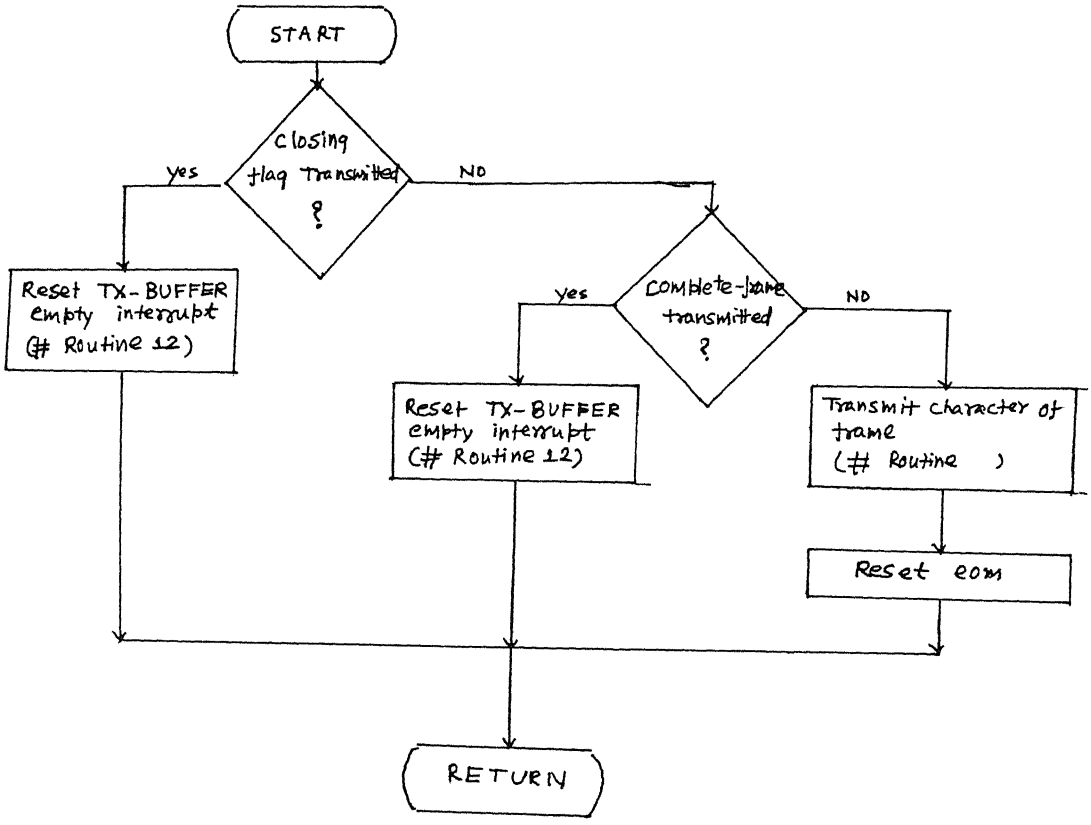


Fig 5.9 Flow chart of TX-ISR routine (# Routine 40)

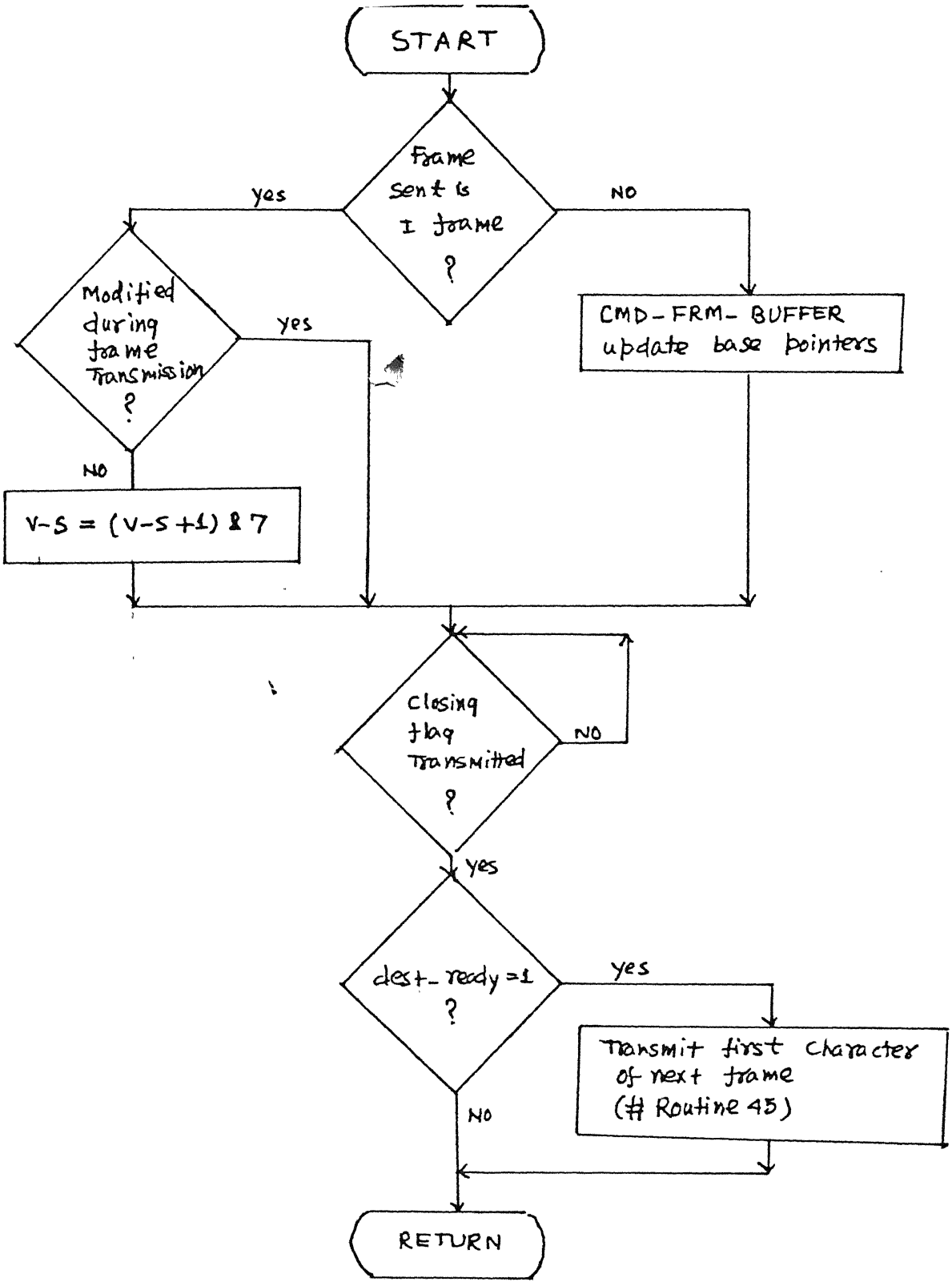


Fig 5.10 Flow chart of eom routine (# Routine 41)

V_S is incremented. However, if V_S has been modified, by data transfer software, during frame transmission, it is not updated. If the frame transmitted is a S-frame or an UN-frame the base pointers of CMD_FRM_BUFFER are updated. After updating these pointers the transmit CRC generator is reset. Then the first character of the next frame,if available, is loaded into the transmit buffer of MPSC-8274 to enable the transmit buffer empty interrupt .

3. Character received : This invokes the 'RX_ISR' routine and the flow-chart of this routine is shown in Fig.5.11. Each received character is loaded into the RX_FRM_BUFFER. Characters of long frames are discarded to prevent the corruption of already received frames. The received characters are also discarded if RX_FRM_BUFFER is full. The CRC bytes are received like normal data bytes. On receiving closing flag the end of frame interrupt occurs.

4. End of frame : This invokes the 'EOF' routine. The flow-chart of this ISR is shown in Fig.5.12. In this ISR, the received frames are discarded if there is a CRC error, or frame is too long, or RX_FRM_BUFFER is full. When a frame is accepted the RX_FRM_BUFFER pointers are updated.

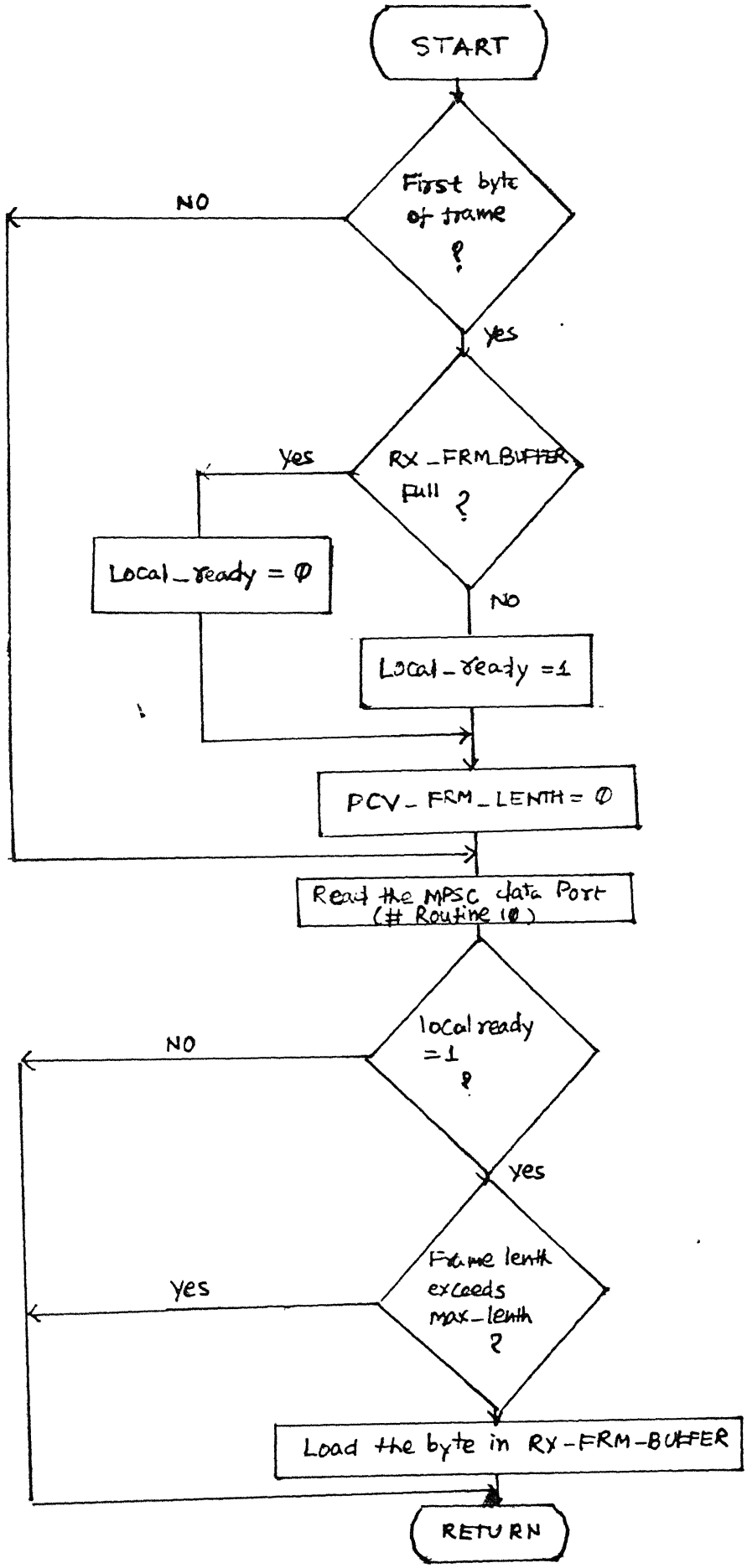


Fig 5.11 Flow chart of RX-TSR routine

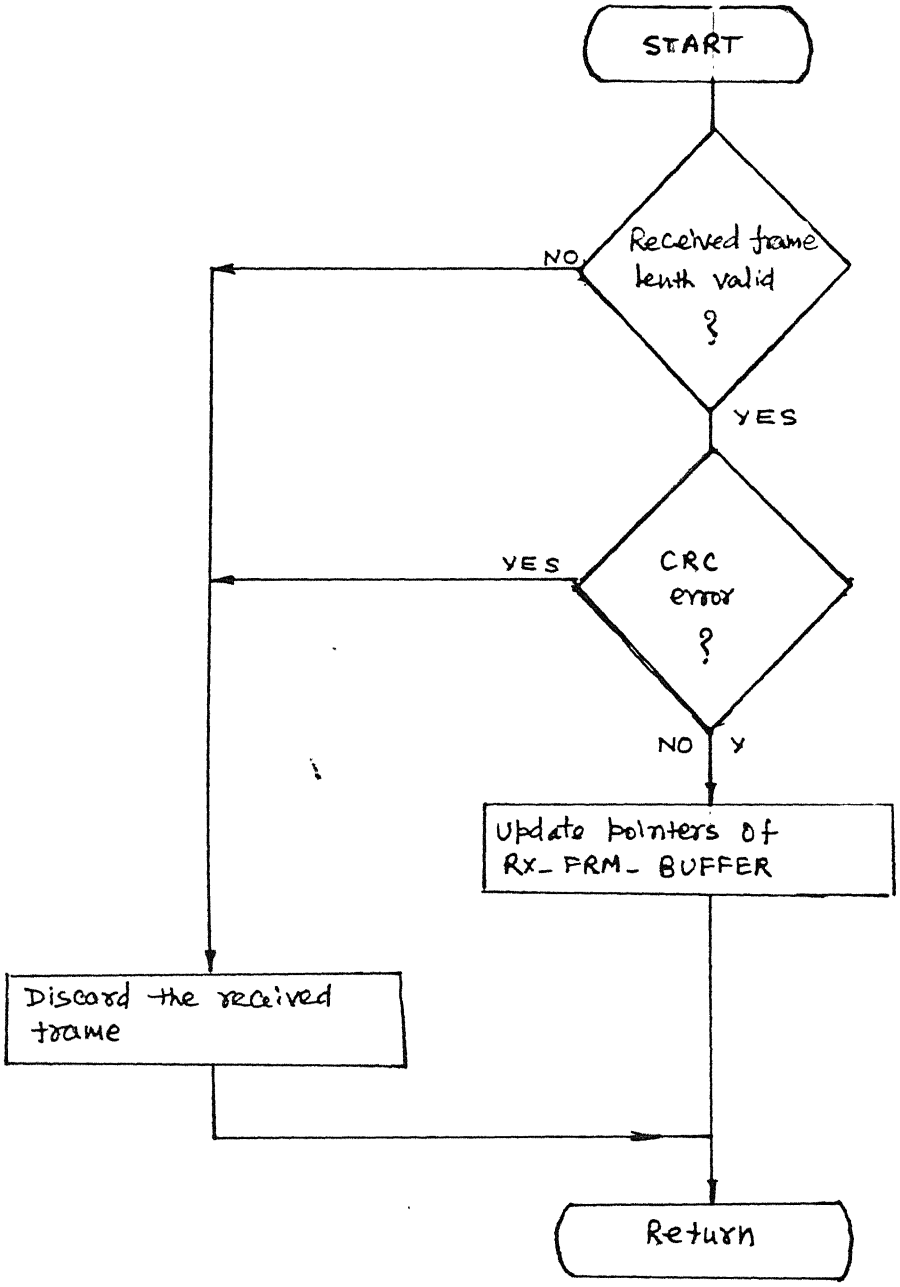


Fig. 5.12 Flow Chart of EOF_FRAME routine (# Routine 43)

5. MPSC-Errors : The errors like receive overrun, and reception of aborted frame are reset by issuing error reset, or external status reset command to the MPSC.

6. PC I/O Port : This invokes the 'PC_INT' ISR. In this ISR the PC I/O port is read and the character is transferred to TX_PKT_BUFFER. Before reading the character the card busy bit (CARD_BUSY), of PC I/O status port, is set if TX_PKT_BUFFER is nearly full. The PC will not write another character to PC I/O port till the CARD_BUSY is reset. The CARD_BUSY is reset, by data transfer routine, when there is sufficient buffer space in the TX_PKT_BUFFER.

CHAPTER 6

CONCLUSIONS

6.1 CONCLUSIONS

The present work is intended to develop an X.25 interface for the IBM PC. The lower two layers i.e., the Physical layer and the Data link layer (LAPB), are implemented and tested in loop back mode. The Physical layer is implemented by the hardware. The hardware is designed to be a general purpose data communication hardware. It supports two serial I/O channels, of which one is used as X.21 port. These channels can be configured in various modes, as explained in section 6.2. The data link layer (i.e., LAPB) is implemented by the hardware and firmware. A major part of the firmware is developed in 'C' language. The firmware, other than I/O interface routines and interrupt service routines, is independent of the hardware used. The major implemented features of LAPB are:

1. All frame types, other than SARM and FRMR, are implemented. Though SARM is defined in LAPB it's usage is not encouraged.
2. Piggybacking of acknowledgments onto I frames.
3. Transmitting acknowledgments with in a specified time period (SEND_ACK_TIME).
4. Time out recovery.

5. Exception recovery for out of sequence errors in reception/transmission of frames.
6. Provision for usage of P/F bits.
7. Link establishment/disconnection.

6.2 SUGGESTIONS FOR FURTHER WORK

1. The RS232-C drivers/receivers on board limit the serial I/O data transfer rate to 19.2 Kbps. To increase this rate the RS-422A drivers/receivers (equivalent of X.27) are to be used.
2. Enhancing the LAPB firmware to implement the following features :
   i. FRMR frame type
   ii. Maximum frame retransmission count.
3. Development of X.25 packet layer: This requires the augmenting of present firmware and development of the higher layer interface software, on the PC.
4. Development of the appropriate transport protocol software, on the PC, to provide various features to the user.
5. With the development of appropriate firmware, the board can support any one of the following :
   i. An X.25 port and an interface to synchronous/asynchronous ring networks

ii. A Packet assembler/disassembler, with an X.25 port and an asynchronous serial I/O connection to host computer/terminal.

APPENDIX A

FIRMWARE LISTING

```
        name    asmisr
_text   segment byte public 'code'
dgroup  group   _data,_bss
        assume  cs:_text,ds:dgroup,ss:dgroup
_text   ends
_data   segment word public 'data'
_d@     label   byte

_data   ends
_bss    segment word public 'bss'
_b@     label   byte

      extrn      _read_char  : byte
      extrn      _write_char : byte
      extrn      _crc_error  :byte
      extrn      _rx_char    :byte
      extrn      _tx_char    :byte
      extrn      _rx_end_frm_base :word
      extrn      _current_time    :word
      extrn      _tx_stat         :byte

_bss    ends

_text   segment byte public 'code'
_main2   proc   near                    # ROUTINE 1
          jmp asm_start
              db 'starting point'
asm_start:      sub ax,ax
                mov bx,01fffh
                mov sp,bx
          assume ss:_bss
                mov ss,ax
;/* Intialize Seg. Reg.s and stack*/
                call _init_vect_table;
                mov ax,0040h
                mov ds,ax
                mov  byte ptr dgroup:_write_char,00
                call near ptr _read_port
;/*Reset Spurious Interrupts*/
                call near ptr _init_timer
                call near ptr _LAPB;
;/* Transfer Control to Data transfer software*/
_asm_end:       jmp short _asm_end
_main2  endp

_init_timer proc near                   # Routine 2
;/*Intialize Timer, Counters in Mode3*/
        push es
        push di
        push bx
        mov bx,1800h
        mov es,bx
        sub bx,bx
        mov di,0003h
```

```
        mov byte ptr es:[di],36h
        mov byte ptr es:[bx],07ch
        mov byte ptr es:[bx],00h    ;/*Ctr0 9.6 khz*/
        inc bx
        mov byte ptr es:[di],76h
        mov byte ptr es:[bx],0c2h
        inc bx
        mov byte ptr es:[bx],00h   ;/*ctr1 10 hz*/
        mov byte ptr es:[di],0b6h
        mov byte ptr es:[bx],0ffh
;/* Ctr2 counts from 0 to ffff h at the rate one count
;   per 0.1 sec*/
end_timer: pop bx
           pop di
           pop es

          push ax
          push di
          push si
          push bp
          push ds
          push es
          jmp  near ptr begin_mpsc
;-------------------------------------------------------------

     ch_a db 00h,18h    ;reset
          db 00h,80h    ;reset TX CRC
          db 00,40h     ;reset rx CRC
          db 02h,30h    ;vectored INT,RTS-B,8086 MODE in
                        ; CH-A and vector in channel B
          db 04h,20h    ;X1CLK,SDLC mode
          db 07h,7eh    ; flag
          db 05h,0ebh   ;TX CRC ENABLE,TX ENABLE,CCITT CRC
                        ;DTR,RTS SET
          db 06h,55h    ; address byte
          db 03h,0d9h   ;RX,RX CRC ENABLE, 8bits/CHAR
          db 01h,1fh   ; 1fh;all int enabled
          db 0ffh       ;EOF
;-------------------------------------------------------------

begin_mpsc:
           mov ax,cs
           mov ss,ax
           mov ax,02800h
           mov ds,ax
           lea bp,ch_a
           mov di,02
           mov si,03
 ch_int:   mov al,[bp]
           cmp al,0ffh
           jz ch_ovr
           mov [di],al
           nop
           nop
```

```
            nop
            nop
            nop
            nop
            nop
            nop
            mov [si],al
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            inc bp
            jmp ch_int
;/*  both channels intialized */
ch_ovr:
          nop
          nop
          nop
          nop
          nop
          nop
          mov byte ptr [si],01
          nop
          nop
          nop
          nop
          nop
          nop
          nop
          nop
          mov byte ptr [si],04h
          nop
          nop
          nop
          nop
          nop
          nop
          nop
          nop
          nop
          nop
          nop
          nop
          nop

          nop
          nop
          nop
          nop
          nop
```

```
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            mov byte ptr  [di],010h
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            mov byte ptr  [si],010h
;/* reset spurious interrupts                    */
               nop
               nop
               nop
               nop
               nop
               nop
               nop
               nop
               nop
               nop
               nop
               nop
               mov  byte ptr [si],30h
               nop
               nop
               nop
               nop
               nop
               nop
               nop
               nop
               nop
               nop
               nop
               nop
               nop
```

```
          mov byte ptr [di],30h
          sub ax,ax
          assume ss:_bss
          mov ss,ax
end_mpsc:    pop es
             pop ds
             pop bp
             pop si
             pop di
             pop ax
             ret
_init_timer endp

_init_mpsc proc near                    # Routine 3
      nop
      ret
_init_mpsc endp

_init_vect_table proc near              # Routine 4

;/*Intialize Vector table*/
                mov ax,0000
                mov ds,ax
                mov dx,cs
                mov bx,(34h*4);
                lea ax,_tx_int
                mov [bx],ax
                add bx,02
                mov [bx],dx
                add bx,02
                lea ax,_ext_status_int
                mov [bx],ax
                add bx,02
                mov [bx],dx
                add bx,02;
                lea ax,_rx_int
                mov [bx],ax
                add bx,02
                mov [bx],dx
                add bx,02
                lea ax,_spl_rx_int
                mov [bx],ax
                add bx,02
                mov [bx],dx
                mov bx,1020
                lea ax,_pc_int
                mov [bx],ax
                add bx,02
                mov [bx],dx
                ret
_init_vect_table endp

_rx_int proc near                       # Routine 5
;/*Receive interrupt ISR, invokes RX_ISR*/
       push ax
```

```
        push bx
        push cx
        push dx
        push si
        push di
        push bp
        push ds
        call near ptr _rx_isr
        call near ptr _eoi    ;issue eoi to 8274
        pop ds
        pop bp
        pop di
        pop si
        pop dx
        pop cx
        pop bx
        pop ax
        iret
_rx_int endp

_tx_int proc near                       # Routine 6
;/*TX buffer empty interrupt ISR, invokes TX_ISR*/
       push ax
       push bx
       push cx
       push dx
       push si
       push di
       push bp
       push ds
       call near ptr _tx_isr
       call near ptr _eoi
       pop ds
       pop bp
       pop di
       pop si
       pop dx
       pop cx
       pop bx
       pop ax
       iret
_tx_int endp

_spl_rx_int proc near                   # Routine 7
;/* 8274 special receive interrupts like RX over-flow*/
          push ax
          push bx
          push cx
          push dx
          push si
          push di
          push bp
          push ds
          push es
          mov ax,2800h
```

```
            mov es,ax
            mov BYTE PTR es:2,01
            mov al,es:2  ;/* read RR1 to find source of INT*/
            mov ah,al
            and al,80h
            jz _spl_ahd2
;/* End of FRAME Interrupt*/
            mov byte ptr dgroup:_crc_error,0
            and ah,40h
            jz _spl_ahd1 ; /*Check for CRC error  */
            mov byte ptr dgroup:_crc_error,1
_spl_ahd1 : call near ptr _eof_frame
              ; Invoke END of FRAME routine*/
_spl_ahd2 : call near ptr _error_reset
              ;/*Issue error reset*/

_r_ahd: pop es
        pop ds
        pop bp
        pop di
        pop si
        pop dx
        pop cx
        pop bx
        pop ax
        call near ptr _eoi
        iret
_spl_rx_int endp

_ext_status_int proc near            # Routine 8

;/*Extenal status interrupts ISR*/

        push ax
        push bx
        push cx
        push dx
        push si
        push di
        push bp
        push ds
        push es
            mov ax,2800h
            mov es,ax
            mov BYTE PTR  es:2,0
            mov al,es:2
            mov ah,al     ; read RR2
            and al,40h
            jz _ext_ahd1
;/* TX under-run*/

            call near ptr _eom
; /* Invoke EOM routine*/
            jmp near ptr _ext_ahd2;
_ext_ahd1: and ah,80h
```

```
            jz _ext_ahd2
;/*Received an aborted frame*/
            call near ptr _abort_frame
_ext_ahd2: call near ptr _reset_ext_status
        pop es
        pop ds
        pop bp
        pop di
        pop si
        pop dx
        pop cx
        pop bx
        pop ax
        call near ptr _eoi
        sti
        iret
_ext_status_int endp

_PC_INT PROC NEAR                       # Routine 9
;/*PC I/O port interrupt ISR*/
    CALL NEAR PTR _READ_PORT
    STI
    RET
_PC_INT ENDP

_IN_PORT PROC NEAR                      # Routine 10

;/*Read Channel A data register*/

            push es
            push ax
            mov ax,2800h
            mov es,ax
            mov al,es:0
            mov ah,00
            mov dgroup:_rx_char,al
            pop ax
            pop es
            ret
_IN_PORT ENDP

_OUT_PORT PROC NEAR                     # Routine 11

;/*Write to channel A data register*/

            PUSH ES
            PUSH AX
            MOV AX,2800H
            MOV ES,AX
            MOV AL,dgroup:_TX_CHAR
            MOV ES:0,AL
            POP AX
            POP ES
            RET
_OUT_PORT ENDP
```

```
_RESET_TX_INT PROC NEAR                    # Routine 12

;/* Reset TX buffer empty interrupt of channel A*/

            PUSH ES
            PUSH AX
            MOV AX,2800H
            MOV ES,AX
            MOV AL,00101000B ;RESET TX INTERRUPT
            MOV ES:2,AL
            POP AX
            POP ES
            RET
_RESET_TX_INT ENDP

_RESET_EOM PROC NEAR                       # Routine 13

;/* Reset End of Message bit of Channel A*/

            PUSH ES
            PUSH AX
            MOV AX,2800H
            MOV ES,AX
            MOV AL,11000000B ;RESET TX INTERRUPT
            MOV ES:2,AL
            POP AX
            POP ES
            RET
_RESET_EOM ENDP


_RESET_TX_CRC PROC NEAR                    # Routine 14

;/* Reset Channel A TX CRC generator*/

            PUSH ES
            PUSH AX
            MOV AX,2800H
            MOV ES,AX
            MOV AL,10000000B
            MOV ES:2,AL
            POP AX
            POP ES
            RET
_RESET_TX_CRC ENDP

_RESET_RX_CRC PROC NEAR                    # Routine 15

;/*Reset Channel A RX CRC generator*/

            PUSH ES
            PUSH AX
            MOV AX,2800H
            MOV ES,AX
```

```
            MOV AL,01000000B
            MOV ES:2,AL
            POP AX
            POP ES
            RET

_RESET_RX_CRC ENDP

_ERROR_RESET PROC NEAR                  # Routine 16

;/*Issue Error-reset command to channel A*/

            PUSH ES
            PUSH AX
            MOV AX,2800H
            MOV ES,AX
            MOV AL,00110000B
            MOV ES:2,AL
            POP AX
            POP ES
            RET
_ERROR_RESET ENDP

_RESET_EXT_STATUS PROC NEAR             # Routine 17

/*Reset Extenal-Status command to Channel A*/

            PUSH ES
            PUSH AX
            MOV AX,2800H
            MOV ES,AX
            MOV AL,00010000B ;RESET EXT/STATUS INTERRUPT
            MOV ES:2,AL
            POP AX
            POP ES
            RET
_RESET_EXT_STATUS ENDP

_EOI PROC NEAR                          # Routine 18

;/*Issue End of Interrupt command to Channel A*/

    PUSH ES
    PUSH AX
    MOV AX,2800H
    MOV ES,AX
    MOV AL,38h;
    MOV ES:2,AL ; Write to WR0
    POP AX
    POP ES
    RET
_EOI ENDP
```

```
_READ_TIMER proc near                    Routine 19

;/* Read Counter2 of 8253/8254 and update time*/

    PUSH ES
    PUSH AX
    MOV AX,1800H
    MOV ES,AX
    MOV AL,ES:2
    MOV AH,ES:2
    mov dgroup:_CURRENT_TIME,AX
    POP AX
    POP ES
    RET
_READ_TIMER ENDP


_ENABLE_INTERRUPT PROC NEAR          # Routine 20
          STI
          RET
_ENABLE_INTERRUPT ENDP


_READ_PORT PROC NEAR                 # Routine 21

;/*Read PC I/O port*/

   PUSH ES
   PUSH AX
   MOV AX,0800H
   MOV ES,AX
   MOV AL,ES:0
   MOV AH,00
   MOV  dgroup:_READ_CHAR,AL
   POP AX
   POP ES
   RET
_READ_PORT ENDP


_WRITE_PORT PROC NEAR                # Routine 22

;/*Write to PC I/O port*/

   PUSH ES
   PUSH AX
   MOV AX,0800H
   MOV ES,AX
   MOV AL,dgroup:_WRITE_CHAR
   MOV ES:0,AL
   POP AX
   POP ES
   RET
```

```
_WRITE_PORT ENDP

_SEND_BUSY PROC NEAR                    # Routine 23

;/*Set CARD BUSY bit of PC I/O status port*/

    PUSH ES
    PUSH AX
    MOV AX,02800H
    MOV ES,AX
    MOV BYTE PTR ES:2,5
    MOV BYTE PTR ES:2,69H
    POP AX
    POP ES
    RET
_SEND_BUSY ENDP

_READ_STATUS_PORT PROC NEAR          Routine 24

;/* Read PC I/O Status Port*/

      PUSH AX
      PUSH ES
      MOV AX,1000H
      MOV ES,AX
      MOV AL,ES:[0]
      MOV AH,AL
      AND AL,020H
      JNZ  STATUS_AHD1
      MOV BYTE PTR DGROUP:_tx_stat,0
      JMP SHORT STATUS_EXIT

STATUS_AHD1:

          MOV BYTE PTR DGROUP:_TX_stat,1

STATUS_EXIT:
      POP ES
      POP AX
      RET
_READ_STATUS_PORT ENDP

_TEXT ENDS

_text   segment byte public 'code'

        public _main2
        extrn  _tx_isr :near
        extrn  _rx_isr :near
        extrn  _lapb   :near
        EXTRN  _EOF_FRAME  :NEAR
        EXTRN   _EOM       :NEAR
        EXTRN   _ABORT_FRAME  :NEAR
        EXTRN   _READ_PC_PORT : NEAR
        public  _write_port
```

X.25 SOFTWARE

```
/* This software implements the LAPB protocol*/
/* The object file of this software is to be linked with
   the object file of X25LIB using the microsoft LINKER.
   The COMMAND to be given for linking is
 > LINK X25LIB + X-25,X-25;
   The X-25.HEX,containing the hex code of the software, can
   be  obtained by deleting the EXE header of the  X-25.EXE,
   output of Linker. The code of X-25.hex is transferred to
   the EPROM and the following bytes are  added, at the
   address locations 0ffff0-0ffff of card :

   OFFFF0 - EA
   OFFFF1 - 00
   OFFFF2 - 00
   OFFFF3 - 00
   OFFFF4 - F8
              Provided an 8k /16k/32k ROM ICs are used*/


/* modified on 8-2-89*/

      #define yes              1
      #define available        1
      #define ready            1
      #define window_size      8      /* Window size of LAPb*/
      #define dest_address     0x0    /* should be 01*/
      #define local_address    0x0    /* should be 03*/
      #define src_address      1
      #define  pkt_size        512    /*Maximum Packet Size*/
      #define max_frm_length   32    /* Max. I field Size*/
      #define data_fr          1
      #define  max_time        0xffff
      #define  rcv_ack_time_out        30    /* 3 sec.*/
      #define   send_ack_time_out       15     /*   send   ack
                   within 1.5 sec. of the receipt  I-frame*/

unsigned char rx_control,n_r,n_s,v_r,v_s,prev_n_r;
unsigned char rx_frm_length,reject_state,dest_ready,cnt;
unsigned char reject_stat,card_busy;
unsigned char address,cmd,cmd_length;
unsigned int  rx_begin_frm_base,rx_begin_frm_pointer;
unsigned int  rx_end_frm_base,rx_end_frm_pointer;
unsigned int  tx_begin_frm_base,tx_begin_frm_pointer;
unsigned int  tx_end_frm_base,tx_end_frm_pointer;
unsigned int  cmd_begin_frm_base,cmd_end_frm_base;
unsigned int  cmd_begin_frm_pointer,cmd_end_frm_pointer;
unsigned int  tx_begin_pkt_base,tx_end_pkt_base;
```

```
unsigned int  tx_end_pkt_pointer,tx_begin_pkt_pointer;
unsigned int  rx_begin_pkt_base,rx_end_pkt_base;
unsigned int  rx_end_pkt_pointer,rx_begin_pkt_pointer;
unsigned int  tx_base,tx_pointer;
unsigned char rx_frm_buffer[8][48];
unsigned char tx_frm_buffer[8][48];
unsigned char pkt_buffer[pkt_size];
unsigned char tx_pkt_buffer[pkt_size];
unsigned char cmd_frm_buffer[8][48];
unsigned int  time_out = 0;
unsigned int  snd_frame = 0;
unsigned int  rcv_frame = 1;
unsigned int  pkt_length = 0;
unsigned int  current_time,send_ack_time,rcv_ack_time[8];
unsigned char local_ready;
unsigned int  temp_word;
unsigned char temp_byte,temp_byte1,temp_byte2;
unsigned char tx_int,frm_sent,data_avlbl,eof_rcv;
unsigned char rcv_data_enable,frm_length,txd_length;
unsigned char TX_CHAR,RX_CHAR,crnt,crnt1;
unsigned char cm_b[20];
unsigned char cm_e[20];
unsigned char cmdp = 0;
unsigned char rx_pkt_free,ack_validity,reject_sent_stat;
unsigned char rx_p_bit,tx_p_bit;
unsigned char lapb_link_set,disc_sent,sabm_sent;
unsigned char loop_count; /*junk*/
unsigned char temp1;
unsigned int  temp2;
unsigned char write_char,read_char,send_ack_set,crc_error;
unsigned char frm_cnt,close_flag_pending;
unsigned char data_ok,tx_stat;
/*****************************************/
/*     LAPB protocol implementation      */
/*****************************************/
main()
{
     MAIN2();
}
lapb()                          # Routine 25
{
begin:
       v_r = 0;prev_n_r = 7;v_s = 0;n_r = 0;n_s =0;
       reject_stat = 0;current_time = 0;local_ready = 0;
       dest_ready =0;send_ack_time = max_time;temp_word =0;
       temp_byte = 0;
       tx_begin_pkt_pointer = 0;tx_end_pkt_pointer = 0;
       rx_begin_pkt_pointer =0;rx_end_pkt_pointer = 0;
       tx_begin_frm_base =0;tx_begin_frm_pointer =0;
       tx_end_frm_base = 0;tx_end_frm_pointer = 0;
       rx_begin_frm_base = 0;rx_begin_frm_pointer = 0;
       rx_end_frm_base = 0;rx_end_frm_pointer = 0;
       cmd_begin_frm_base = 0;cmd_end_frm_base = 0;
       cmd_end_frm_pointer = 0;cmd_begin_frm_pointer = 0;
       tx_base = 0;
```

```
        tx_pointer = 0;
        txd_length = 0;

        cnt = 0;
        crnt = 0;
        rx_begin_frm_base     = 0;
        temp_word = 0;
        local_ready = 1;
        dest_ready  = 1;
        data_avlbl    = ~yes;
        frm_sent      = data_fr;
        eof_rcv       = yes;
        tx_int        = 0;
        rcv_data_enable = 1;/*Varables Intialized*/
        frm_length = 0;
        temp_byte = 20;
        send_ack_time = max_time;
        crnt1    = 0;
        crnt     = 0;
        rx_begin_frm_base = 0;
        rx_end_frm_base   = 0;
        cmd_begin_frm_base  = 0;
        pkt_length          = 0;
        rx_pkt_free         = 1;
        rx_p_bit            = 0;
        tx_p_bit            = 0;
        reject_sent_stat    = 0;
        temp_word           = 0;
        card_busy           = 0;

      while (temp_word < 8)
      {
      rcv_ack_time[temp_word] = max_time;++temp_word;
      }/* Intialize receive ack timers*/
/*_____________________________________________________*/

/*     while (tx_end_pkt_pointer < (pkt_size - 1))
      {
      tx_pkt_buffer[tx_end_pkt_pointer] = 32;++tx_end_pkt_pointer;
      }*/
/*     while (rx_end_pkt_pointer < (pkt_size - 1))
      {
      pkt_buffer[rx_end_pkt_pointer] = 0;++rx_end_pkt_pointer;
      }
      rx_end_pkt_pointer = 0;*/
/*_____________________________________________________*/

     lapb_link_set = 1;/*should be 0*/
     disc_sent = 0;
     sabm_sent = 0;
     loop_count = 0;/*junk*/
     crc_error  = 0;
     send_ack_set = 1;
     frm_cnt      =  0;
     close_flag_pending = 0;
```

```
/*Intialization completed , program begins*/


    enable_interrupt();

/*    send_sabm();
    send_first_char();*/  /* Comment out in local loopback*/

lapb_start: read_timer();
/*_____________________________________________________________________*/
if (rx_begin_pkt_pointer != rx_end_pkt_pointer)
 {
 read_status_port();
 if (tx_stat == 0)
  {
   write_char = pkt_buffer[rx_begin_pkt_pointer];
   write_port();
   rx_begin_pkt_pointer = (rx_begin_pkt_pointer + 1) & (pkt_size - 1);

  }
 }    /* send data to pc if avaible*/
/*_____________________________________________________________________*/

  if (send_ack_time <= current_time )
       { /* Send ack time out send ack*/
       if ( check_cmd_frm_buffer() == 1 )
          {/* load RR or RNR into command buffer*/
           send_ack_time  = max_time;
           cmd_frm_buffer[cmd_end_frm_base][0] = 0;
           cmd_frm_buffer[cmd_end_frm_base][1] = dest_address;
           if ( local_ready == 1 )
            { /* load RR */
             cmd_frm_buffer[cmd_end_frm_base][2] = ((v_r << 5) | 1);
            }
           else
            {    /* Load RNR*/
             cmd_frm_buffer[cmd_end_frm_base][2] = ((v_r << 5) | 5);
             }
                temp_word = ((cmd_end_frm_base +1) & 07);
                cmd_end_frm_base = temp_word;
                reset_send_ack_timer();
                }

       }
    if (current_time >= rcv_ack_time[n_r])
       {
        temp_word = 0;
        while (temp_word < 8)
        {
        rcv_ack_time[temp_word] = max_time;temp_word++;
        }
        v_s = n_r; /* Receive ack failed*/
       }
```

```
rxfr:
        if ( rx_begin_frm_base != rx_end_frm_base )
         { /* Received frame in Rx_FRM_BUFFER*/
          receive_frame();
         }
       if (rx_p_bit == 1 )
        { /* A command frame with P bit set to be acknowledged*/
        if ( check_cmd_frm_buffer() == 1 )
            (/* load RR or RNR into command buffer*/
             rx_p_bit = 0;
             cmd_frm_buffer[cmd_end_frm_base][0] = 0;
             cmd_frm_buffer[cmd_end_frm_base][1] = local_address;
             if ( local_ready == 1 )
             { /* load RR , with F bit set*/
             cmd_frm_buffer[cmd_end_frm_base][2] = ((v_r << 5) | 0x11);
             }
             else
             {     /* Load RNR with F bit set*/
             cmd_frm_buffer[cmd_end_frm_base][2] = ((v_r << 5) | 0x15);
             }
                  temp_word = ((cmd_end_frm_base +1) & 07);
                  cmd_end_frm_base = temp_word;
                  reset_send_ack_timer();/* Acknowledgement sent*/
          }
        }
txfr: if (lapb_link_set != yes) goto lapb_start;
       if (tx_begin_pkt_pointer != tx_end_pkt_pointer)
        {  /*tx pkt has data*/
          if (((tx_end_frm_base + 1) & 7) != tx_begin_frm_base)
           {
          send_frame();  /* Formulate I frames */
           }
        }
     else
        {
/*          -- tx_end_pkt_pointer; */ /*_______JUNK__________*/
        }

       if ((dest_ready == 1) & (tx_int != yes))
        {  /* If tx interrupt is reset and there is  data  to
              send  enable tx interrupt and load new frame*/
       send_first_char();
       }
if((local_ready != yes)&(((rx_end_frm_base+1)&7)!=rx_begin_frm_base))
         {
          if (check_cmd_frm_buffer() == yes)
           (/* Send RR to indicate ready state*/
           local_ready = yes;
            cmd_frm_buffer[cmd_end_frm_base][0] = 0;
            cmd_frm_buffer[cmd_end_frm_base][1] = dest_address;
            cmd_frm_buffer[cmd_end_frm_base][2] = ((v_r << 5) | 1);
            cmd_end_frm_base    = (cmd_end_frm_base + 1) & 7;
            reset_send_ack_timer();
           }
```

```
        }
f ((local_ready==yes)&(((rx_end_frm_base+1)&7)==rx_begin_frm_base))
         {/*Send RNR to indicate not ready state*/
         if (check_cmd_frm_buffer() == yes)
          {
          local_ready = 0;
           cmd_frm_buffer[cmd_end_frm_base][0] = 0;
           cmd_frm_buffer[cmd_end_frm_base][1] = address;
           cmd_frm_buffer[cmd_end_frm_base][2] = ((v_r << 5) | 5);
           cmd_end_frm_base  = (cmd_end_frm_base + 1) & 7;
           reset_send_ack_timer();
           }
         }

:_____________________________________________________*/
       data_ok = 0;
     if (rx_pkt_free != 1)
      {
      temp2 = 0;
      while (temp2 < 480)
       {
    if (pkt_buffer[temp_byte] != tx_pkt_buffer[temp_byte]) goto lapb_end
       ++temp2;

       }
       data_ok = 1;
      }
pb_end: if (data_ok == 1)
         {
          write_char = 0x55;
          write_port();
          write_char = 0xaa;
          write_port();
         }*/
_____________________________________________________*/


     if (rx_pkt_free != 1)
         {
          write_char = 55;
          write_port();*/
         }
*************For LoopBack only**************************/
       dest_ready = local_ready;
*****************************************************/
       ack_validity =  1;
       goto lapb_start;


kt_main()


ceive_frame()                    # Routine 26

  read_frm_control_info();/* read control field of recv. frame*/
```

```
if ((rx_control & 0x10) != 0)
      {
if (rx_frm_buffer[rx_begin_frm_base][1] == dest_address) tx_p_bit = 0;
/* recevied frame with F bit sent*/
if (rx_frm_buffer[rx_begin_frm_base][1] == local_address) rx_p_bit = 1;
/*received frame with P bit set */
      }
     if ((rx_control & 0x03) == 0x03)
        { /*Received frame is UN frame*/
         analyze_un_frame();
        }
     else
        {
         if (lapb_link_set != yes)
          {
           if (check_cmd_frm_buffer() == 1)
            { /* If link in disconnected state send DM*/
             send_dm();
             inc_rx_begin_frm_base();return(0);
            }
          }
        n_r = (rx_control & 0xe0) >> 0x5;
        if (ack_valid() == yes)
        {
        if ((reject_stat == yes) & (n_r == prev_n_r)) goto rcv_exit1;
        reject_stat = 0;
        release_tx_buffer(); /*release tx_buffer from prev n(r)
                                                to present n(r)*/
        }
rcv_exit1:if ((rx_control & 0x01) != 0)
            {
              analyze_s_frame();
              inc_rx_begin_frm_base();
            }
        else
           {/* If space is available in packet buffer
               read the frame  and load in rx_packet buffer*/
           /* if packet buffer full exit*/
           n_s = rx_control & 0x0e;
           n_s = n_s >>1;
           if (n_s == v_r)
             {
             read_length();
             if (rx_frm_length < 40)
             {
             if (rx_pkt_free != 1)
             {
             return(0) ;
             }
             reject_sent_stat = 0;
             v_r = ((v_r + 1) & 7); /*Advance recv. sequence no.*/
             wr_pkt_buffer();      /*Transfer information to
                                     RX_PKT_BUFFER        */
             pkt_free();          /* Is RX_PKT_BUFFER full*/
             set_send_ack_timer();/* Start send ack. timer*/
```

```
            }
  rx_exit: inc_rx_begin_base();

          }
           else
             {
/* Send Reject*/
/* If command buffer not free return*/

             if (reject_sent_stat != 1)
              {
               if (check_cmd_frm_buffer() != yes)
                {
                    return(0);
                }
         send_reject(); /* If space available in command frm buffer*/
         reject_sent_stat = 1;
               }
         inc_rx_begin_frm_base();
            }
          }
       }
   }
 analyze_s_frame()              # Routine 27
   {
    int short temp;
       temp = ((rx_control >>2) & 03);
          ++temp;
          switch(temp)
          { /* dtermine the frame type*/
           case 1:
                   dest_ready = 1;/* Received RR*/
                   break;
           case 2: dest_ready = 0;/*Received RNR*/
                  break;
           case 3: if (reject_stat != yes)  /* Received reject*/
                    {/* Accept REJ if no recv. REJ is outsanding*/
                     if (ack_validity == 1)
                      {
                           reject_stat = yes;
                           v_s = n_r;
                       }
                    }
                    break;
         default: break;
         }
}
read_length()                  # Routine 28
     {
        rx_frm_length = rx_frm_buffer[rx_begin_frm_base][0];
        }
read_frm_control_info()        # Routine 29
     {
        rx_control = rx_frm_buffer[rx_begin_frm_base][2];
     }
```

```
release_tx_buffer()                    # Routine 30
   {/*    Discard   acknowledged   frames   from   transmit    window
                                                        TX_FRM_BUFFER*/

          temp1 = n_r;

          temp1 = temp1 + 8; /*mod8 window*/
          temp1 = temp1 - prev_n_r;
          temp1 = temp1 & 7;
          temp2 = tx_begin_frm_base;
          while (temp1 > 0)
               {
                   temp2 = (++temp2) & (window_size - 1);--temp1;
                }
         tx_begin_frm_base =temp2;
         temp1 =prev_n_r;
         while (temp1 != n_r)
         {
         rcv_ack_time[temp1] = -1;
         temp1 = (temp1 + 1) & 7 ;
         }
         prev_n_r = n_r;
        }
send_reject()                          # Routine 31
{/*   Formulate   reject   frame   and   send   it   if   command   buffer
CMD_FRM_BUFFER is not full*/
/*If cmd_buffer not_free return*/
address = dest_address;
cmd = ((v_r << 5) | 9);
if (rx_p_bit == 1)
 {
  rx_p_bit = 0;
  cmd = cmd | 0x10;
  address = local_address;/*____________*/

  }

cmd_length =0;
cmd_frm_buffer[cmd_end_frm_base][0] =cmd_length;
cmd_frm_buffer[cmd_end_frm_base][1] = address;
cmd_frm_buffer[cmd_end_frm_base][2] = cmd;
cmd_end_frm_base = ((cmd_end_frm_base + 1) & 7);
reset_send_ack_timer();
}

send_frame()                           # Routine 32
{Formulate I frame and load it in in transmit window*/
tx_frm_buffer[tx_end_frm_base][1] = dest_address;
tx_frm_buffer[tx_end_frm_base][2]=(((tx_end_frm_base<<1)|(v_r<<5))&0xee)
tx_end_frm_pointer = 3;
temp_byte =0;
while((temp_byte<max_frm_length)
                                   &(tx_begin_pkt_pointer!=tx_end_pkt_pointer
{
tx_frm_buffer[tx_end_frm_base][tx_end_frm_pointer]
```

```
                                        =tx_pkt_buffer[tx_begin_pkt_pointer];
tx_begin_pkt_pointer = (tx_begin_pkt_pointer + 1) & (pkt_size-1);
++tx_end_frm_pointer;++temp_byte;
}
tx_frm_buffer[tx_end_frm_base][0] = temp_byte;
/*rcv_ack_time[tx_end_frm_base] = current_time + rcv_ack_time_out;*/
temp_word        = tx_end_frm_base;
tx_end_frm_base = (++temp_word) & (window_size-1);
reset_send_ack_timer();
}

inc_rx_begin_base()              # Routine 33
{
temp2 = rx_begin_frm_base;
rx_begin_frm_base = (++temp2) & 7;
}
wr_pkt_buffer()                  # Routine 34
{ /* Write I field contents to RX_PKT_BUFFER*/
  rx_begin_frm_pointer = 2;
  while (rx_frm_length > 0)
        {
pkt_buffer[rx_end_pkt_pointer] =
               rx_frm_buffer[rx_begin_frm_base][++rx_begin_frm_pointer]
        --rx_frm_length;
      rx_end_pkt_pointer = (rx_end_pkt_pointer  + 1) & (pkt_size - 1);
        }
}
pkt_free()                       # Routine 35
{ /*Is RX_PKT_BUFFER full*/
  temp2 = (rx_begin_pkt_pointer + pkt_size);
  temp2 = ((temp2 - rx_end_pkt_pointer) & (pkt_size -1));
  if (temp2 > max_frm_length)
  {
  rx_pkt_free = 1;
   return(0);
  }
  rx_pkt_free = 0;
  return (0);
  }
  check_cmd_frm_buffer()         # Routine 36
  { /* Check if CMD_FRM_BUFFER is full*/
   if(((cmd_end_frm_base+1)&7)!=cmd_begin_frm_base) return(yes);
   return(~yes);
  }
  no_pkt_data()                  # Routine 37
   {/* Is there data in TX_PKT_BUFFER, for serial transmission*/
   if (tx_begin_pkt_pointer == tx_end_pkt_pointer) return(yes);
   return(~yes);
   }

ack_valid()                      # Routine 38
  { /* Is the acknowledgement valid*/
   temp_byte1 = ((n_r + 8) - prev_n_r) & 7 ;
   temp_byte2 = prev_n_r;
   while (temp_byte1 > 0)
```

```
                {
        temp_byte2= (++temp_byte2) & 7 ;--temp_byte1;
                  if (temp_byte2 == v_s) goto ack_invalid_exit;
                  }
                   ack_validity = 1;
                   return(yes);
ack_invalid_exit: ack_validity = 0;return(~yes);
   }

inc_rx_begin_frm_base()        # Routine 39
{
temp_word = rx_begin_frm_base;
rx_begin_frm_base = (++temp_word) & 7;
}

/*____________________ISR routines_____________________________________

tx_isr()                       # Routine 40
{
       if (close_flag_pending == 1)
        {
        reset_tx_int();
        close_flag_pending = 0;/*Closing flag to be transmitted*/
        return(0);
        }

       if (tx_pointer <=  txd_length)
           {
        if (frm_sent != data_fr)
         {
           TX_CHAR = (cmd_frm_buffer[tx_base][tx_pointer]);
           OUT_PORT();
           reset_eom();

         }
        else
         {
           TX_CHAR = (tx_frm_buffer[tx_base][tx_pointer]);
           OUT_PORT();
           reset_eom();

         }   /*transmit frame contents*/
           ++tx_pointer;
           }
        else
           {
         RESET_TX_INT();/* After transmitting last byte of frame rese
                                TX_INT*/
       close_flag_pending  =  1;/* Closing flag to  be  transmitted b
                                                        8274*/
/*         tx_int = ~yes;*/
/*         eom();for 8250 only, delete when 8274 is used*/
           }
}
```

```
eom()                                   # Routine 41
{
          enable_interrupt();

/*        reset_eom();*/
          if (frm_sent == data_fr)
           {
if((tx_frm_buffer[tx_base][1] == dest_address)
             & ((tx_frm_buffer[tx_base][2] & 0x10) != 0)) tx_p_bit = 1
           if (tx_base == v_s )
            {
            v_s = ((v_s + 1) & 7);
            }
          }
      else
           {
if((cmd_frm_buffer[tx_base][1]==dest_address)
           &((cmd_frm_buffer[tx_base][2] & 0x10) != 0)) tx_p_bit = 1;
           cmd_begin_frm_base =( cmd_begin_frm_base + 1) & 7;
       } /* Upadate pointers for next frame transmission*/
    while (close_flag_pending == 1)
     {
     }/* Wait till closing flag is sent*/

     if (dest_ready == 1)
      {
       send_first_char();
      } /*Intiate frame transmission*/
     else
        {
       data_avlbl = ~yes;
        }
}

rx_isr()                                # Routine 42
{
  if (eof_rcv == yes)
        {
          rx_end_frm_pointer = 1;
          IN_PORT();
/*        frm_length         = ( RX_CHAR + 2 );for 8250*/
          eof_rcv            = ~yes;
          if (((rx_end_frm_base + 1) & 7) != rx_begin_frm_base)
           {
             rcv_data_enable = 1;
/*      rx_frm_buffer[rx_end_frm_base][rx_end_frm_pointer] = RX_CHAR;
              ++rx_end_frm_pointer;*/
            return(0);
/*ON  receiving first character of frame update pointers
                                    and neglect first character*/
           }
       else
           {
                rcv_data_enable = 0;
                local_ready     = ~yes;
```

```
                    return(0);
/* If receive frm buffer full set local ready to not ready*/
           }
         }
           if (rx_end_frm_pointer <= (max_frm_length + 4 ))
            {/*If frame size within limits accept the received characte
              , otherwise ignore it */
               IN_PORT();
            rx_frm_buffer[rx_end_frm_base][rx_end_frm_pointer] = RX_CHAR;
               ++rx_end_frm_pointer;
            }
         else
            {
                     IN_PORT();
                     ++rx_end_frm_pointer;
             }
/*           if (rx_end_frm_pointer > frm_length)
           {
           eof_frame() ;for 8250 only
           }                                        */

}
eof_frame()                        # Routine 43
{/* If closing flag is received, check validity of frame and update
    pointers if the frame is valid*/
  eof_rcv      = 1;
  if ((rx_end_frm_pointer <= (max_frm_length + 5))
                                                & (rcv_data_enable ==1))
   {
    if (crc_error == 0)
     {
    if (rx_end_frm_pointer > 4)
     {

    rx_frm_buffer[rx_end_frm_base][0] = (rx_end_frm_pointer - 4);
    rx_end_frm_base                   = (rx_end_frm_base + 1) & 7;
    ++crnt;
     }

     }
    }

}
abort_frame()                      # Routine 44
{/* Received an aborted frame*/
   eof_rcv = 1;
}

send_first_char()                  # Routine 45
{/* Send first character of frame if destination is ready
    there is a frame to be transmitted
    non-I frames have higher priority in transmission
  */

    reset_tx_crc();
```

```
     if (tx_p_bit != 1)
      {
         if (cmd_begin_frm_base == cmd_end_frm_base)
         {
xyz:     if (v_s != tx_end_frm_base)
          {
          frm_sent = data_fr;
          tx_base  = v_s;
          data_avlbl = 1;
          tx_pointer = 1;
          TX_CHAR = tx_frm_buffer[tx_base][tx_pointer];
          txd_length = (tx_frm_buffer[tx_base][0] + 2);tx_int = yes;
          ++tx_pointer;++crnt1;
          rcv_ack_time[v_s] = current_time + rcv_ack_time_out;

          OUT_PORT();
          reset_eom();

/*          en_tx_int();*/    /*_________for 8250 only___________*/
          }
          else
                {
                  data_avlbl = 0;
                  tx_int     = 0;
                  return(0);

                 }
         }
      else
         {
             frm_sent = ~data_fr;
             tx_base  = cmd_begin_frm_base;
             tx_pointer = 1;
             data_avlbl = 1;
             TX_CHAR = cmd_frm_buffer[tx_base][tx_pointer];
              ++tx_pointer;tx_int =  yes;
             txd_length = (cmd_frm_buffer[tx_base][0] + 2);out_port();
             reset_eom();

        }


    }
}
reset_send_ack_timer()          # Routine 46
{
  send_ack_time = max_time;
  send_ack_set  = 0;

}
analyze_un_frame()              # Routine 47
          { Analyze received UN frame*/
          goto ua_end;/*_____________________*/
```

```
        if (((rx_control & 0xef) == 0x2f) & (address == local_address))
         {                                                    /*local_address*
          sabm();return(0);
         }
        if (((rx_control & 0xef) == 0x43) & (address == local_address))
         {                                                    /*local_addres
          disc();return(0);
         }
        if ((rx_control & 0xef) == 0x63) /*& (address == local_address)
         {
          ua();return(0);                                  /*dest_address*/
         }
        if (((rx_control & 0xef) == 0xf)  & (address == dest_address))
         {
          dm();return(0);
         }
        if (((rx_control  & 0xef) == 0x87) & (address == dest_address))
         {/*frmr received*/
         }
_end:   inc_rx_begin_frm_base();
        }
    sabm()                      # Routine 48
        {
         /*abort_tx_frm();*/
         /*abort_pc();*/
         reset_tx_int();
         v_r = 0;v_s = 0;n_s = 0;n_r = 0;prev_n_r = 0;
         tx_begin_frm_base = 0;
         tx_end_frm_base   = 0;
         cmd_begin_frm_base = 0;
         cmd_end_frm_base   = 0;
         tx_int = ~yes;
         txd_length    = 0;
         local_ready   = 1;
         dest_ready    = 1;
         data_avlbl    =  ~yes;
         frm_sent      = data_fr;
         /*rcv_data_enable = 0;*/
         if (check_cmd_frm_buffer() == yes )
          {
          send_ua(); /* Ack. received SABM*/
          lapb_link_set = yes;inc_rx_begin_frm_base();
          }
        }
    disc()                      # Routine 49
        {
         if (check_cmd_frm_buffer() == yes)
          {
          send_ua();
          lapb_link_set = ~yes;inc_rx_begin_frm_base();
          }
        }

     ua()                       # Routine 50
        {
```

```
          inc_rx_begin_frm_base();
          if (sabm_sent == 1)
           {
            sabm_sent = 0;lapb_link_set = yes;dest_ready = 1;return(
           }
          if (disc_sent == 1)
           {
           disc_sent = 0;lapb_link_set  = 0; dest_ready = 0;return(0
           }
           dest_ready = 1;
        }
     dm()                        # Routine 51
        {
        dest_ready = 0;lapb_link_set = 0;inc_rx_begin_frm_base();
        }
 send_ua()                       # Routine 52
        {return(0);/*__________junk_________*/
        cmd_frm_buffer[cmd_end_frm_base][0] = 0;
        cmd_frm_buffer[cmd_end_frm_base][1] = local_address;
        cmd = 0x63;
        if (rx_p_bit == 1) cmd = (cmd | 0x10);;
        cmd_frm_buffer[cmd_end_frm_base][2] = cmd;
        temp_word = cmd_end_frm_base;
        cmd_end_frm_base = ((++temp_word) & 7);
        }
send_sabm()                      # Routine 53
       {
        sabm_sent = 1;
        lapb_link_set = 0;
        cmd_frm_buffer[cmd_end_frm_base][0] = 0;
        cmd_frm_buffer[cmd_end_frm_base][1] = dest_address;
        cmd_frm_buffer[cmd_end_frm_base][2] = 0x2f;/*|0x10*/
        temp_word = cmd_end_frm_base;
        cmd_end_frm_base = ((++temp_word) & 7);
       }
send_disc()                      #Routine 54
       {
        disc_sent = 1;
        cmd_frm_buffer[cmd_end_frm_base][0] = 0;
        cmd_frm_buffer[cmd_end_frm_base][1] = dest_address;
        cmd_frm_buffer[cmd_end_frm_base][2] = 0x43;/*|0x10*/
        temp_word = cmd_end_frm_base;
        cmd_end_frm_base = ((++temp_word) & 7) ;
       }
 send_dm()                       #Routine 55
        {return(0);
     cmd_frm_buffer[cmd_end_frm_base][0] = 0;
        cmd_frm_buffer[cmd_end_frm_base][1] = local_address;
     cmd = 0xf;
        if (rx_p_bit == 1)    cmd = cmd | 0x10;
        cmd_frm_buffer[cmd_end_frm_base][2] = cmd;
        temp_word = cmd_end_frm_base;
        cmd_end_frm_base = ((++temp_word) & 7) ;
        }
```

```
set_send_ack_timer()            #Routine 56
{
 if (send_ack_set != 1)
   {
   send_ack_time = current_time + send_ack_time_out;
   send_ack_set = 1;
   }


ead_pc_port()                   # Routine 57

*_____________________________________________________________________*/

*if (((tx_end_pkt_pointer + 3) & (pkt_size - 1)) == tx_begin_pkt_pointer
 {
 send_busy();
 card_busy = 1;

 }*/
 read_port();
 tx_pkt_buffer[tx_end_pkt_pointer] = read_char;
 tx_end_pkt_pointer = ((tx_end_pkt_pointer + 1) & (pkt_size - 1));


t_____________________________________________________________________
```

REFERENCES

[1] Uyless Black, 'Computer Networking Protocols and Standards', Printice-Hall inc., NJ

[2] CCITT Recommendation X.21, 'Interface between DTE and DCE for Synchronous operation on Public Data Networks, 1980'

[3] CCITT Recommendation X.25, 'Interface between DTE and DCE for Terminals operating in the Packet mode on Public Data Networks',1980

[4] Antony Rybczynski, 'X.25 Interface and End to End Virtual circuit service characteristics', IEEE Transactions on Communications Vol. COM-28,No.4, April 1980.

[5] H.V.Bertine, 'Physical Level Protocols', IEEE Transactions on communications Vol.COM-28, No.4, April 1980.

[6] Maj. S.L.Kapoor, 'Design of Workstation based X.25 Protocol Analyzer', M.Tech Thesis, I.I.T Kanpur, 1986.

[7] Microcommunications Handbook (pp. 2-112 to 2-149 and 2-383 to 2-419), Intel, 1988

[8] Microsystem component Data Handbook(Vol. I), (pp.3-106 to 3-133) , Intel, 1988

[9] Technical Reference - Personal Computer PC-XT

[10] B.W.Kenighan & D.M.Ritchie, 'The C Programming Language', PHI

[11] 'TURBO C Reference Guide',Borland

[12] Schildt, 'Advanced TURBO C',Borland Osborne MC_Grawhill

GENERAL REFERENCES

[13] A.S.Tannenbaum, 'Computer Networks', PHI

[14] Harold.C.Folds, 'Procedures for Circuit Switched Service in Synchronous Public Data Networks', IEEE Transactions on Communications Vol. COM-28 No.4,April 1980

[15] Sanjeev Verma, 'Low Cost PC LAN Network', M.Tech Thesis, I.I.T. Kanpur,1988

EE-1989-M-RAV-DES.